# Is JWT Token Really Secure?

Shuai LI

If you are already familiar with the JWTs (JSON Web Token)[5], then you can skip to section 2 where I discuss about the possible attacks and fixes found in [6][7][9]. Section 1 gives an introduction to JWTs and one application example which JWT found really handy - OAuth access tokens.

## I. JWT

All JWTs are constructed from three different elements: the header, the payload and the signature. The first two elements are JSON objects. The third element is the signature on the first two elements. All the elements are encoded using base64 URL encoding algorithm. JWT supports a variety of signature algorithms (HMAC, RSA, ECDSA). Figure 1 shows an example of the JWT.

```
eyJhbGciOiJFUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
leHBpcmVJbiI6IjEgZGF5Iiwic3ViamVjdCI6IjE
wMDAiLCJhdWRpZW5jZSI6Imh0dHA6Ly9sb2NhbGh
vc3Q6NDk5MC9nZXRSZXNvdXJjZSIsImlzc3VlciI
6Imh0dHA6Ly9sb2NhbGhvc3Q6NTAwMC9hdXRob3J
pemF0aW9uIiwib2JqZWN0QXR0cmlidXRlcyI6eyJ
yZXNvdXJjZVR5cGUiOlsiSGVhcnQiXX0sImFjdGl
vbkF0dHJpYnV0ZXMiOnsiYWN0aW9ucyI6WyJ2aWV
3Il19LCJpYXQiOjE1NzA0MDM1Mjd9.IsqSgmNlSA
YGBEhVp6BtxLNjzQXOyXdrBSEB3ufef6kHdDIjVo
4PCcM2m4DiF4otVE3GTuZF66PaV4WKOAEdyQ
```

Fig. 1: JWT example

### A. The Header

The red element in Figure 1 is the header. Figure 2 shows the result after decoding. The main purpose of the header is to specify the token type and the signature algorithm which is used to sign the token. The token in Figure 1 is a jwt which is signed using ECDSA algorithm with curve P-256 and SHA-256[1].

```
{
  "alg": "ES256",
  "typ": "JWT"
}
```

Fig. 2: Header of Token in Figure 1

### B. The Payload

The next element in Figure 1 is the payload. Figure 3 shows the decoded result. The payload is the place where all the access data is added to. Each assertion is called a claim, the developers can add customized claims as they need, with the exceptions of some already registered claims[5].

Some of the registered claims are:

- **iss**: The issuer of this token.
- **sub**: Usually a machine-readable identifier of the client that this token is issued to.
- **aud** : Service-specific string identifier or list of string identifiers representing the intended audience for this token.
- **iat**: Indicating when this token was originally issued.
- **exp** : Indicating when this token will expire.
- **nbf**: Indicating when this token is not to be used before.
- **scope**:A JSON string containing a space-separated list of scopes associated with this token.

```
{
  "expireIn": "1 day",
  "subject": "1000",
  "audience": "http://localhost:4990/getResource",
  "issuer": "http://localhost:5000/authorization",
  "objectAttributes": {
    "resourceType": [
      "Heart"
    ]
  },
  "actionAttributes": {
    "actions": [
      "view"
    ]
  },
  "iat": 1570403527
}
```

Fig. 3: Payload of Token in Figure 1

### C. Signature

The last element of the token is the signature, showing in blue part. The use of the signature is to ensure the authenticity of the token. Only the token generated by the trusted authority is considered valid. The authority, when creating the access token, first encodes the header and the payload using the base64 algorithm, then signs on the concatenation of the encoded data. Finally, the signature is appended to the first two elements. The verifier uses the appropriate key to validate the authenticity of the token when receives a token,

### *D. JWT and OAuth 2.0*

In OAuth protocol[4], the AS returns an access token to the client if the permissions requested by the client are authorized. The access token can be decoded as a JWT[3]. Signed JWTs make good access tokens. The AS can include all the necessary data in the token and sign the token to avoid token introspection[8]. The RS can verify the token and understand the authorization information carried inside of the token without sending the token back to the AS.

## II. Attacking JWTs

### *A. Change the signing algorithm Attack*

**Attack:** For example, if we have a JWT, the decoded header and the payload look like this:

```
header:{
    alg:”ES256”,
    typ:”JWT”
},
payload:{
    sub:”Joe”,
    role:”user”
}
```

Since this is a signed token, everyone is free to see the content. Now the attacker can modify the header and payload to this:

```
header:{
    alg:”none”,
    typ:”JWT”
},
payload:{
    sub:”Joe”,
    role:”admin”
}
```

If the attacker manages to use this token correctly, he or she may escalate the privilege to admin. You probably would point out that this attack would not work since the original signature does not valid anymore on the tampered data. However, this was a severe attack in the past. The reason is that a lot of jwt libraries choose signature algorithm depending on the $alg$ information in the token header. The token verification function usually take two parameters, $token$ and $secret$. To pick the correct signature algorithm, the function relies on the $alg$ claim from the header. In the example above, since the attacker modified the $alg$ to $none$, this means that there is no signature algorithm, the verification will always succeed regardless the presence of a valid signature.

**Fix:** Many libraries today report ”alg”: ”none” token as invalid. The other mitigation is to require the verification algorithm as an additional input to the verification function, rather than replying on the $alg$ claim.

### *B. Changing the algorithm from RS256 to HS256 Attack*

**Attack:** This attack exploits the same vulnerability - verification function relies on the $alg$ claim from the header to pick the signature algorithms. Now, suppose the attacker gets an access token signed with an RSA key pair.

```
header:{
    alg:”RS256”,
    typ:”JWT”
},
payload:{
    sub:”Joe”,
    role:”user”
}
```

The token is signed with a RSA private key. Everyone else in the world should be able to verify the signature using the corresponding public key. The attacker can forge a new token using the following scheme. First, he or she modifies the header and the payload to,

```
header:{
    alg:”HS256”,
    typ:”JWT”
},
payload:{
    sub:”Joe”,
    role:”admin”
}
```

The algorithm ”HS256” stands for HMAC using SHA 256. Then, the attacker generates a new signature using the public key as the secret key fed to HMAC. Let’s take a look at the verification function, the function will take two parameters, $tokem$ and $publickey$. But now, instead of using the public key to verify the RSA signatures, verification function uses it as the shared secret for the HS256 algorithms. The verification function will accept the forged token as a valid token. Now the attacker can use the forged token to access the resources with escalated access.

**Fix:** The mitigation against this attack is similar to the ones in Section II-A.

### *C. Invalid Elliptic-Curve Attack*

**Attack:**In elliptic-curve cryptography, the public key is a point on the elliptic curve, while the private key is simply a number that sits within a special, but very big, range. Some implementations, fail to validate the inputs to any arithmetic operations on elliptic curves. If inputs to these operations are not validated, the arithmetic operations may produce seemingly valid results even when they are not. These results, when used in the context of cryptographic operations such as decryption, can be used to recover the private key[2]. This attack has been demonstrated in the past[9]. This class of attacks are known as invalid curve attacks. Good-quality implementations always verify that public-keys are a valid elliptic-curve point for the chosen curve and that private keys sit inside the valid range of values.

### *D. Substitution Attack- Different Recipient*

**Attack:** This attack is possible when $aud$ claim is missing in the JWTs. Consider the case of one AS, multiple resource servers. All resource servers should be able to validate the token from the AS. If a token does not contain $aud$ information, the malicious client can use the token on the other resource servers that have the same AS. The client may attain unauthorized permissions from a unintended RS since the token, after all, has a valid signature. In the example below, the client can use this token to attain admin role in a different RS.

```
payload:{
    sub:"Joe",
    role:"admin"
}
```

**Fix:** To prevent this attack, the token must include an $aud$ claim uniquely specifying the intended audience.

### *E. Substitution Attack- Same Recipient*

**Attack:** In the above example, we showed a token can be used to access different resource servers who share the same AS. The solution is to include an unique $aud$ claim in every token. In practise, one RS may provide different services, for example, the company resource server has two different databases, company-resource/item-database and company-resource/user-database. The access to each database has different policies. If AS generates a token for client A and only grants the access to the item-database, the payload of the token could look like this,

```
payload:{
    sub:"Joe",
    role:"admin"
    sub:"company-resource/item-database"
}
```

However, when validating the token, the validation function made a mistake: It did not validate the $aud$ correctly. Instead of checking for an exact match, the function checked for the presence of the company-resource string. An attacker can leverage this and access to the user-database using the token issued for item-database.

**Fix:** To prevent this attack, the validation function must check for the exact match of the $aud$ claim.

## References

[1] A. B. Association et al. Public key cryptography for the financial services industry: The elliptic curve digital signature algorithm (ecdsa). *ANSI X9*, pages 62–1998.

[2] I. Biehl, B. Meyer, and V. Müller. Differential fault attacks on elliptic curve cryptosystems. In *Annual International Cryptology Conference*, pages 131–146. Springer, 2000.

[3] O. W. Group. The oauth 2.0 authorization framework: Jwt secured authorization request (jar). https://tools.ietf.org/html/draft-ietf-oauth-jwsreq-19, 2019. Work in progress. Accessed on May 2019.

[4] D. Hardt. The oauth 2.0 authorization framework. Technical report, 2012. https://tools.ietf.org/html/rfc6749.

[5] M. Jones, P. Tarjan, Y. Goland, N. Sakimura, J. Bradley, J. Panzer, and D. Balfanz. Json web token (jwt). Technical report, 2012. https://tools.ietf.org/html/rfc7519.

[6] S. Langkemper. Attacking jwt authentication. https://www.sjoerdlangkemper.nl/2016/09/28/attacking-jwt-authentication/, 2016. Accessed on Oct 2019.

[7] S. Peyrott. The jwt handbook. *Seattle, WA, United States*, 2016.

[8] J. Richer. Oauth 2.0 token introspection. Technical report, 2015.

[9] A. Sanso. Critical vulnerability uncovered in json encryption. http://blogs.adobe.com/security/2017/03/critical-vulnerability-uncovered-in-json-encryption.html. Accessed on Oct 2019.