Emin Muhammadi

8 Dec, 2021
Github: https://github.com/eminmuhammadi/applied-crypto-project

Applied cryptography project seminar

1. Introduction

JSON Web Token (JWT) is an open standard that defines a compact and self-contained way for securely transmitting information between parties as a JSON object. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a secret (with the HMAC algorithm) or a public/private key pair using RSA or ECDSA.

The aim of the project is directed to implementation of the JWT standard (JSON Web Tokens RFC7519) using RSA (RS256, RS384, RS512) and ECDSA (ES256, ES384, ES512) on Authentication server to generate access tokens for registered users. The access tokens will be used on the Application server to check whether the user has been already registered to use resources or not. The advantage of that standard is that only the Authentication server can access to read the user's details, and it will be only executed on the token generation process. Application server can verify tokens which are obtained by the Authentication server using the public key of the Authentication server.

Tokens are unique, can be readable by others and only accessible during lifetime. There are two types of tokens: access and refresh tokens. Access tokens can unlock every feature of the Application server but it has an expiration time, and can only be renewed by refresh tokens. Each renewed token will be blacklisted, so intruders only have a short time period to brute force endpoints.

2. Benefits

There are benefits to using JWTs when compared to simple web tokens (SWTs) and Security Assertion Markup Language (SAML) tokens.

- **More compact**: JSON is less verbose than XML, so when it is encoded, a JWT is smaller than a SAML token. This makes JWT a good choice to be passed in HTML and HTTP environments.
- **More secure**: JWTs can use a public/private key pair in the form of an X.509 certificate for signing. A JWT can also be symmetrically signed by a shared secret using the HMAC algorithm. And while SAML tokens can use public/private key pairs like JWT, signing XML with XML Digital Signature without introducing obscure security holes is very difficult when compared to the simplicity of signing JSON.
- **More common**: JSON parsers are common in most programming languages because they map directly to objects. Conversely, XML doesn't have a natural document-to-object mapping. This makes it easier to work with JWT than SAML assertions.
- **Easier to process**: JWT is used at internet scale. This means that it is easier to process on user's devices, especially mobile.

3. Use cases

JWTs can be used in various ways:

- **Authentication**: When a user successfully logs in using their credentials, an ID token is returned. According to the OpenID Connect (OIDC) specs, an ID token is always a JWT.
- **Authorization**: Once a user is successfully logged in, an application may request to access routes, services, or resources (e.g., APIs) on behalf of that user. To do

so, in every request, it must pass an Access Token, which may be in the form of a JWT. Single Sign-on (SSO) widely uses JWT because of the small overhead of the format, and its ability to easily be used across different domains.

- **Information Exchange**: JWTs are a good way of securely transmitting information between parties because they can be signed, which means you can be sure that the senders are who they say they are. Additionally, the structure of a JWT allows you to verify that the content hasn't been tampered with.

4. JSON Web Token Structure

All JWTs have JSON Web Signatures (JWSs), meaning they are signed rather than encrypted. A JWS represents content secured with digital signatures or Message Authentication Codes (MACs) using JSON-based data structures. A well-formed JWT consists of three concatenated Base64url-encoded strings, separated by dots **(.)**:

- **JOSE Header**: contains metadata about the type of token and the cryptographic algorithms used to secure its contents.
- **JWS payload (set of claims):** contains verifiable security statements, such as the identity of the user and the permissions they are allowed. The JWT specification defines seven reserved claims that are not required, but are recommended to allow interoperability with third-party applications. These are:
    - **iss (issuer)**: Issuer of the JWT
    - **sub (subject)**: Subject of the JWT (the user)
    - **aud (audience)**: Recipient for which the JWT is intended
    - **exp (expiration time)**: Time after which the JWT expires
    - **nbf (not before time)**: Time before which the JWT must not be accepted for processing
    - **iat (issued at time)**: Time at which the JWT was issued; can be used to determine age of the JWT
    - **jti (JWT ID)**: Unique identifier; can be used to prevent the JWT from being replayed (allows a token to be used only once

- **JWS signature**: used to validate that the token is trustworthy and has not been tampered with. When you use a JWT, you must check its signature before storing and using it.

Signing Algorithms. Most JWTs in the wild are just signed. The most common algorithms are:

A. **HMAC + SHA256**

Hash-Based Message Authentication Codes (HMACs) are a group of algorithms that provide a way of signing messages by means of a shared key. In the case of HMACs, a cryptographic hash function is used (for instance SHA256). The strength (i.e. how hard it is to forge an HMAC) depends on the hashing algorithm being used.

The main objective in the design of the algorithm was to allow the combination of a key with a message while providing strong guarantees against tampering. Ad-hoc solutions (for example, appending the key to the message and then hashing the result) suffer from mathematical flaws that allow potential attackers to forge the signature. The HMAC algorithm is designed against that.

B. **RSASSA-PKCS1-v1_5 + SHA256**

C. **ECDSA + P-256 + SHA256**

Both RSA and ECDSA are asymmetric encryption and digital signature algorithms. What asymmetric algorithms bring to the table is the possibility of verifying or decrypting a message without being able to create a new one. This is key for certain use cases. Picture a big company where data generated by the sales team needs to be verified by the accounting team. If an HMAC were to be used to sign the data, then both the sales team and the accounting team would need to know the same key. This would allow the sales team to sign data and make it pass as if it were from the accounting team. Although this might seem unlikely, especially in the context of a corporation, there are times when the ability to verify the creator of a signature is essential. JWTs signed or encrypted with RSA or ECDSA provide this capability. A party uses its private party to sign a JWT. Receivers in turn use the public key (which must be shared in the same way as an HMAC shared key) of that party to verify the JWT. The receiving parties cannot create new JWTs using the public key of the sender.

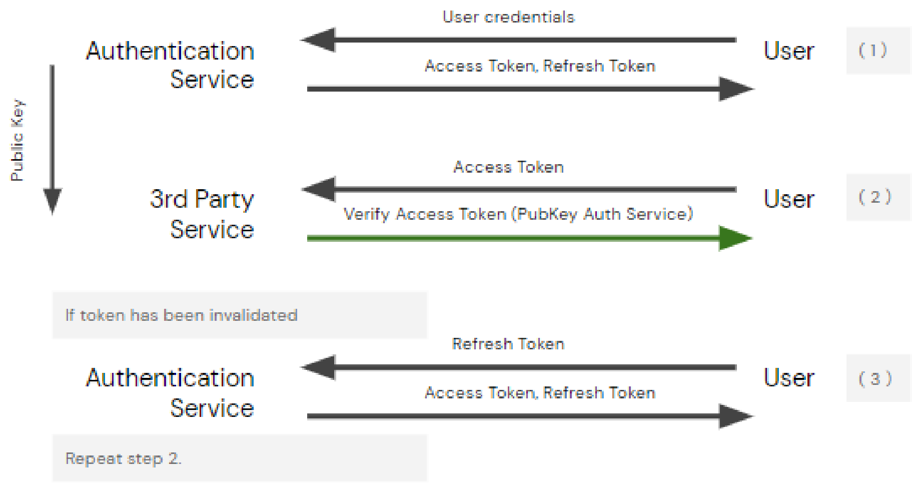
The main difference between RSA and ECDSA lies in speed and key size. ECDSA requires smaller keys to achieve the same level of security as RSA. This makes it a great choice for small JWTs. RSA, however, is usually faster than ECDSA.

Signature. To create the signature part you have to take the encoded header, the encoded payload, a secret, the algorithm specified in the header, and sign that. For example if you want to use the HMAC SHA256 algorithm, the signature will be created in the following way:

```
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), secret)
```

The signature is used to verify the message wasn't changed along the way, and, in the case of tokens signed with a private key, it can also verify that the sender of the JWT is who it says it is.

5. Implementation



Following diagram displays the full communication process between User, Authentication Service and 3rd Party Service (Application server).

Step 1. User sends login credentials to Authentication Service and if user credentials are correct gets two tokens: access token and refresh token. Access token has been intended to establish authorized communication channels between user and 3rd Party Service. There is an expiration time for access token, thus refresh token will automatically regenerate new token pairs. If a token has been renewed, old tokens will be blacklisted by the Authentication Service.
Step 2. In this step the user sends his/her access token to 3rd Party Service to use its features. 3rd Party Service verifies each signature using the public key of the Authentication server.
Step 3. If an access token has been expired or invalidated, 3rd Party Service will decline all requests from the user. It is the responsibility of the Authentication Service to renew new token pairs.

6. Test results

Two tests are conducted to benchmark signing and verification algorithms. Following results display that overall performance for each test case:

```
     http_req_blocked...............: avg=1.23ms   min=0s      med=998.9µs
max=35.72ms p(90)=2ms      p(95)=3.27ms
     http_req_connecting............: avg=1.11ms   min=0s      med=754.6µs
max=35.72ms p(90)=1.85ms   p(95)=3ms
   ✓ http_req_duration..............: avg=4.7s     min=84.2ms  med=1.69s
max=12.04s  p(90)=11.32s   p(95)=11.54s
       { expected_response:true }...: avg=4.7s     min=84.2ms  med=1.69s
max=12.04s  p(90)=11.32s   p(95)=11.54s
     http_req_failed................: 0.00%   ✓ 0          ✗ 3526
     http_req_receiving.............: avg=216.06µs min=0s      med=0s
max=13.36ms p(90)=507.15µs p(95)=859.65µs
     http_req_sending...............: avg=94.34µs  min=0s      med=0s
max=5.5ms   p(90)=503.7µs  p(95)=998.77µs
     http_req_tls_handshaking.......: avg=0s       min=0s      med=0s
max=0s      p(90)=0s       p(95)=0s
     http_req_waiting...............: avg=4.7s     min=83.83ms med=1.69s
max=12.04s  p(90)=11.32s   p(95)=11.54s
     http_reqs......................: 3526    14.676317/s
     iteration_duration.............: avg=10.42s   min=1.27s   med=12.97s
max=14.18s  p(90)=13.36s   p(95)=13.42s
     iterations.....................: 1763    7.338158/s
     vus............................: 2       min=2       max=100
     vus_max.......................: 100     min=100     max=100
```

Load test

```
     http_req_blocked..............: avg=579.39µs min=0s      med=532.2µs
max=1.95ms   p(90)=786.11µs p(95)=1.02ms
     http_req_connecting...........: avg=528.67µs min=0s      med=529.05µs
max=1.95ms   p(90)=648.44µs p(95)=777.8µs
   ✓ http_req_duration.............: avg=144.38ms min=78.44ms med=179.99ms
max=311.58ms p(90)=204.47ms p(95)=209.12ms
       { expected_response:true }...: avg=144.38ms min=78.44ms med=179.99ms
max=311.58ms p(90)=204.47ms p(95)=209.12ms
     http_req_failed...............: 0.00%   ✓ 0         ✗ 370
     http_req_receiving............: avg=121.83µs min=0s      med=0s
max=1.5ms    p(90)=519.71µs p(95)=654.85µs
     http_req_sending..............: avg=47.48µs  min=0s      med=0s
max=997.3µs  p(90)=50.55µs  p(95)=504.83µs
     http_req_tls_handshaking.......: avg=0s       min=0s      med=0s
max=0s       p(90)=0s       p(95)=0s
     http_req_waiting..............: avg=144.21ms min=78.44ms med=179.75ms
max=311.58ms p(90)=204.3ms  p(95)=208.58ms
     http_reqs.....................: 370     1.540924/s
     iteration_duration............: avg=1.29s    min=1.26s   med=1.29s
max=1.42s    p(90)=1.31s    p(95)=1.32s
     iterations....................: 185     0.770462/s
     vus...........................: 1       min=1      max=1
     vus_max.......................: 1       min=1      max=1
```

Smoke test

7. Vulnerabilities (Security)

JSON Web Tokens (JWTs) provide a way to securely exchange data using JSON objects. They are often used in authorization because they can be signed, verified, and therefore trusted – but only if implemented correctly. Here is a technical deep dive into JSON Web Token attacks and vulnerabilities.

Allowing the None algorithm. The JWT standard accepts many different types of algorithms to generate a signature: RSA, HMAC, Elliptic Curve, None. The None algorithm specifies that the token is not signed. If this algorithm is permitted, we can bypass signature checking by changing an existing algorithm to None and stripping the signature.

Algorithm confusion. JWT accepts both symmetric and asymmetric encryption algorithms. When an application uses asymmetric encryption, it can openly publish its public key and keep the private key secret. This allows the application to sign tokens using its private key and anyone can verify this token using its public key. The algorithm confusion vulnerability arises when an application does not check whether the algorithm of the received token matches the expected algorithm.

kid parameter injections. The JWT header can contain the Key Id parameter kid. It is often used to retrieve the key from a database or filesystem. The application verifies the signature using the key obtained through the kid parameter. If the parameter is injectable, it can open the way to signature bypass or even attacks such as RCE, SQLi, and LFI.

Attacks using the jku header. In the JWT header, developers can also use the jku parameter to specify the JSON Web Key Set URL. This parameter indicates where the application can find the JSON Web Key (JWK) used to verify the signature – basically the public key in JSON format. An attacker can change the jku parameter value to point to their own JWK instead of the valid one. If accepted, this allows the attacker to sign malicious tokens using their own private key. After the malicious token is sent, the application will fetch the attacker's JWK and use it to verify the signature.

8. Conclusion

In conclusion, JWTs are a convenient way of representing authentication and authorization claims for your application. They are easy to parse, human readable and compact. But the killer features are in the JWS and JWE specs. With JWS and JWE all claims can be conveniently signed and encrypted, while remaining compact enough to be part of every API call. Solutions such as session-ids and server-side tokens seem old and cumbersome when compared to the power of JWTs.

9. References

- Auth0 Docs - https://auth0.com/docs/
- RFC 7523 - https://datatracker.ietf.org/doc/html/rfc7523

- JSON Web Token attacks and vulnerabilities - https://www.netsparker.com/blog/web-security/json-web-token-jwt-attacks-vulnerabilities/