JWT

JSON Web Token

What is JSON Web Token?

- JWT is an open standard (RFC 7519) 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 **HMAC+**SHA256 algorithm).
  - or a public/private key pair using **RSA** or ECDSA (Elliptic Curve).

- Compact:
  - Because of its size, it can be sent through an URL, POST parameter, or inside an **HTTP header**.
- Self-contained:
  - The payload contains all the required information about the user, to avoid querying the database more than once.

The JSON Web Token Structure

- JSON Web Tokens consist of three parts separated by dots (.), which are:
  - Header
  - Payload
  - Signature

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4
gRG9lIiwiaXNTb2NpYWwiOnRydWV9.
4pcPyMD09olPSyXnrXCjTwXyr4BsezdI1AVTmud2fU4
```

JWT Header (aka JOSE header)

- The header *typically* consists of two parts:
  - the type of the token, which is JWT
  - the hashing algorithm such as HMAC SHA256 or RSA

- For example:

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

- This JSON will be **Base64Url** encoded to form the first part of the JWT.

JWT Payload

- The second part of the token is the payload, which contains the claims.
- Claims are statements about an entity (typically, the user) and additional metadata.
- There are three types of claims: *reserved*, *public*, and *private.*

Reserved claims

- These are a set of predefined claims
  - which are not mandatory but recommended
  - thought to provide a set of useful, interoperable claims
- For example:
  - **iss** (issuer)
  - **exp** (expiration time)
  - **sub** (subject)
  - **aud** (audience)
  - **Iat** (Issued At)
  - **nbf** (Not Before)
  - **jti** (JWT ID)

Public claims

- **These can be defined at will by those using JWTs.**

- But to avoid collisions they should be defined in the IANA JSON Web Token Registry or be defined as a URI that contains a collision resistant namespace.

Private claims

- These are the custom claims created to share information between parties that agree on using them.

- An example of payload:

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true
}
```

- This JSON will be **Base64Url** encoded to form the second part of the JWT

Signature

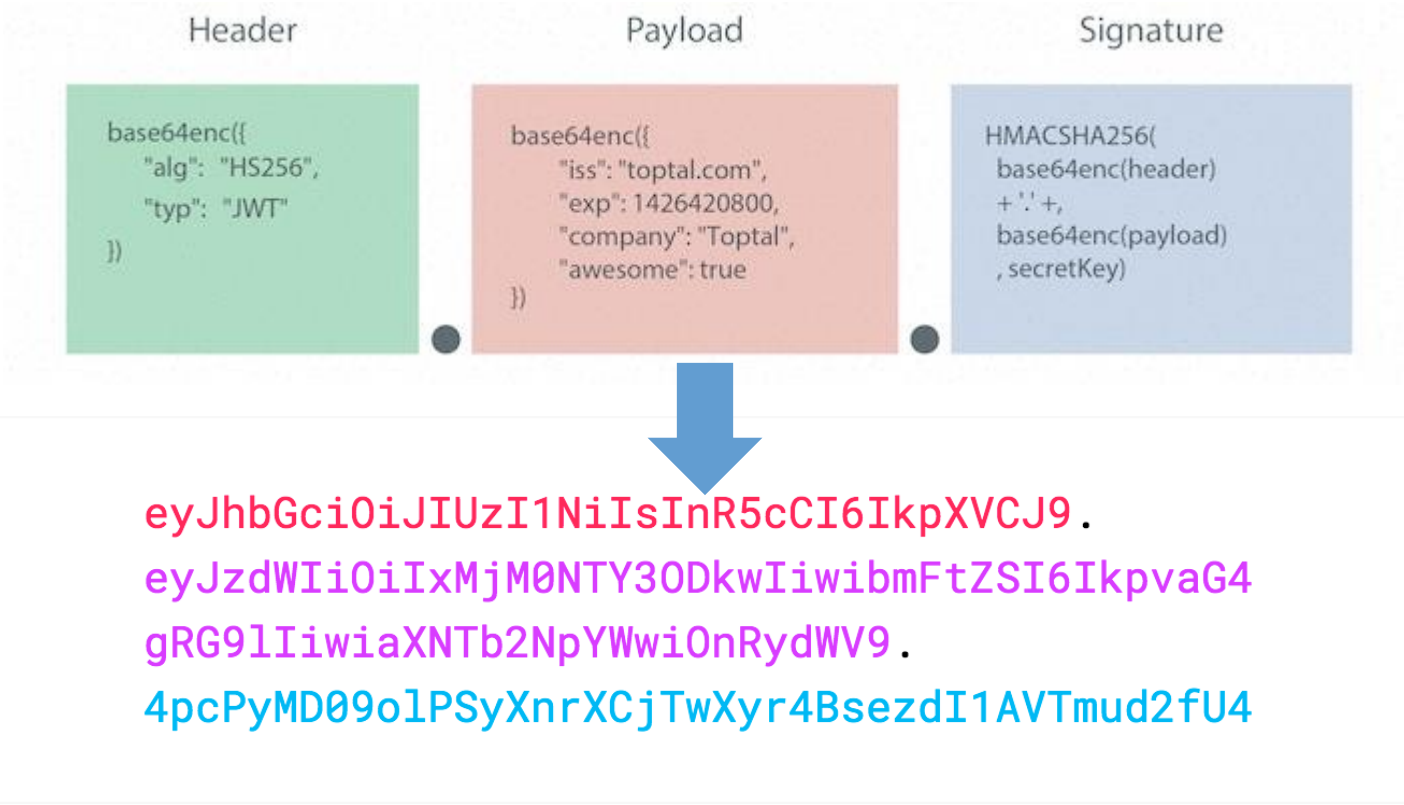
- To create the signature part you have to:
  - take the encoded header
  - the encoded payload
  - a secret
  - the algorithm specified in the header
- And sign that.
- If you want to use the HMAC SHA256 algorithm, the signature will be created in the following way

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

- The signature is used to verify that the sender of the JWT is who it says it is and to ensure that the message wasn't changed in the way.

The Output JWT

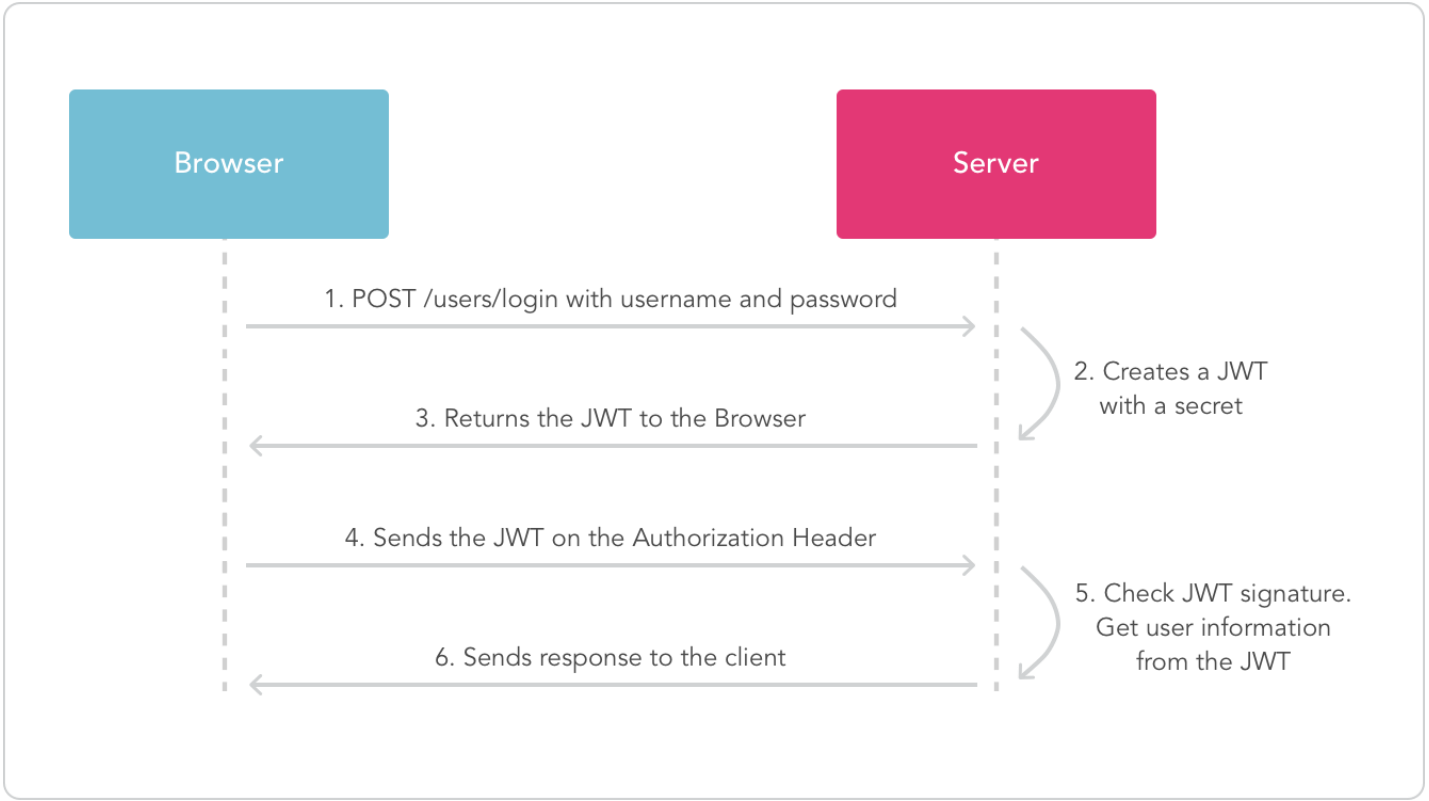
- The output is three Base64 strings separated by dots that can be easily passed in HTML and HTTP environments.
- The previous header and payload encoded, and it is signed with a secret.



How do JSON Web Tokens work?

- The user successfully logs in using his credentials, a JSON Web Token will be returned and must be saved locally (typically in local storage).
- Whenever the user wants to access a protected route or resource, the App sends the JWT, typically in the **Authorization** header using the **Bearer** schema:

```
Authorization: Bearer <token>
```



JWS vs JWE

- A JWT is usually complemented with a signature or encryption.
- These are handled in their own specs as:
  - JSON Web Signature (JWS)
    - allows a JWT to be *validated* against modifications.
  - JSON Web Encryption (JWE)
    - makes sure the content of the JWT is only readable by certain parties.

- Most JWTs are just signed.
- The most common algorithms are:
  - HMAC + SHA256
    - the most common algorithm for signed JWTs
  - RSASSA-PKCS1-v1_5 + SHA256
  - ECDSA + P-256 + SHA256

} Asymetric with private-public key pair

JWT Vulnerabilities

- Some libraries have critical vulnerabilities allowing attackers to bypass the verification step!
- What's wrong?

Let's Try To Verify A Token

- First, we need to determine what algorithm was used to generate the signature.
- No problem, there's an alg field in the header that tells us just that.

- But wait, we haven't validated this token yet, which means that we haven't validated the header!
- This puts us in an awkward position:
  - **in order to validate the token, we have to allow attackers to select which method we use to verify the signature ☹**

The "none" Algorithm

- The *none* algorithm is a curious addition to JWT.
- It is intended to be used for situations where the integrity of the token has already been verified.
- It is one of only two algorithms that are mandatory to implement
  - the other being HS256.
- Some libraries treated tokens signed with the *none* algorithm as a valid token with a verified signature.

- The result?
  - Anyone can create their own "signed" tokens with whatever payload they want, allowing arbitrary account access on some systems.

The "none" Algorithm Check

- Most implementations now have a basic check to prevent this attack:
  - if a secret key was provided, then token verification will fail for tokens using the none algorithm

```
verify(clientToken, serverHMACSecretKey)
```

Asymmetric Signing Algorithms

- The JWT spec also defines a number of asymmetric signing algorithms (based on RSA or ECDSA).

- With these algorithms, tokens are created and signed using a private key, but verified using a corresponding public key.

- If you publish the public key but keep the private key to yourself, only you can sign tokens, but anyone can check if a given token is correctly signed.

JWT Library Implementations

- Typical library implementation

```
// sometimes called "decode"
verify(string token, string verificationKey)
// returns payload if valid token, else throws an error
```

- In systems using HMAC signatures, verificationKey will be the server's secret signing key

```
verify(clientToken, serverHMACSecretKey)
```

- In systems using an asymmetric algorithm, verificationKey will be the public key

```
verify(clientToken, serverRSAPublicKey)
```

Unfortunately, an attacker can abuse this!

Abuse of asymmetric signing algorithms

- If a server is expecting a token signed with RSA, but actually receives a token signed with HMAC, **it will think the public key is actually an HMAC secret key**

```
verify(clientToken, serverRSAPublicKey)
```

- **Hackers how-to:**
  - grab your favourite JWT library, and choose a payload for your token
  - get the public key used on the server as a verification key
  - sign your token using the public key as an HMAC key

```
forgedToken = sign(tokenPayload, 'HS256', serverRSAPublicKey)
```

Recommendation

- The server and client should already know what algorithm was used to sign tokens, and it's not safe to allow attackers to provide this value
  - **Never use the algorithm specified in the header of JWT!**

```
verify(string token,
       string algorithm,
       string verificationKey)
```

References & Links

- JSON Web Tokens
  http://jwt.io/
- Epoch & Unix Timestamp Conversion Tools
  https://www.epochconverter.com/
- Learn all about the different JWT signing algorithms and how to choose the correct one for your use case
  https://auth0.com/blog/json-web-token-signing-algorithms-overview/
- RFC 7519: JSON Web Token (JWT)
  https://tools.ietf.org/html/rfc7519
- RFC 7518: Cryptographic Algorithms for Digital Signatures and MACs
  https://tools.ietf.org/html/rfc7518#section-3
- Critical vulnerabilities
  https://auth0.com/blog/2015/03/31/critical-vulnerabilities-in-json-web-token-libraries/
- JSON Web Token (JWT) Implementation for .NET
  https://github.com/jwt-dotnet/jwt