Never Trust The Client

In this lesson, we'll see how JWTs can be used to prevent clients from tampering with data.

We'll cover the following

As we've seen before, cookies that are issued by our servers can be tampered with, especially if they're not `HttpOnly` and are accessible by JS code on your page.

At the same time, even if your cookies are `HttpOnly`, storing plaintext data in them is not secure, as any client (even `curl`), could get a hold of those cookies, modify them and re-issue a request with a modified version of the original cookie.

Suppose your session cookie contains this information:

```
profile=dXNlcm5hbWU9TGVCcm9uLHJvbGU9dXNlcg==;
```

The string is *base64-encoded*, and anyone could reverse it to get to its actual value, `username=LeBron,role=user`. Anyone could, at that point, replace `user` with `admin` and re-encode the string, altering the value of the cookie.

If your system trusts this cookie without any additional check, you're in trouble. You should never trust the client and prevent them from being able to easily tamper with the data you've handed off. A popular workaround to this issue is to encrypt or sign this data, like JSON Web Tokens do.

JSON Web Tokens

Let's drift for a second and dive into JWT, as their simplicity lets us understand the security mechanism behind them extremely well. A JWT is made of three parts: **headers, claims, and signatures**, separated by a *dot*.

```
JWT = "$HEADER.$CLAIMS.$SIGNATURE"
```

Each value is base64-encoded, with headers and claims being nothing but an encoded JSON object.

```
$HEADER = BASE64({
  "alg": "HS256",  # HMAC SHA 256
  "typ": "JWT"     # type of the token
})

$CLAIMS = BASE64({
  "sub": "1234567890", # ID of the user
  "name": "John Doe",  # Other attributes...
  "iat": 1516239022    # issued at
})

JWT = "$HEADER.$CLAIMS.$SIGNATURE"
```

The last part, the signature, is the Message Authentication Code (abbr. MAC) of the combined `$HEADER.$CLAIM`, calculated through the algorithm specified in the header itself (`HMAC SHA-256` in our case). Once the MAC is calculated, it is base64-encoded as well:

```
$HEADER = BASE64({
  "alg": "HS256",
  "typ": "JWT"
})

$CLAIMS = BASE64({
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
})

$SIGNATURE = BASE64(HS256("$HEADER.$CLAIMS", $PRIVATE_KEY))

JWT = "$HEADER.$CLAIMS.$SIGNATURE"
```

E-voila, our JWT is here!

If you have followed us this far, you have understood that JWT is simply composed of three parts: two insecure sets of strings and a signed one, which is what we use to verify the authenticity of the token. Without the signature, JWTs would be insecure and (arguably) useless, as the information they contain is simply base64-encoded.

As a practical example, let's take a look at this token.

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikpva
G4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
```

As you can see, we have three base64-encoded strings, separated by dots. Reversing them in bash is straightforward.

```
$ cut -d'.' -f1 <<< $TOKEN | base64 -d
{"alg":"HS256","typ":"JWT"}
$ cut -d'.' -f2 <<< $TOKEN | base64 -d
{"sub":"1234567890","name":"John Doe","iat":1516239022}
```

As you would expect, the signature produces garbage instead.

```
$ cut -d'.' -f3 <<< $TOKEN | base64 -d
I▯J▯□IHNJ(]▯O▯▯▯lj□~N▯%base64: invalid input
```

That's the mechanism JWTs use to prevent clients from tampering with the tokens themselves. When a server validates a token, it will first verify its signature through the public key associated with the private one used to generate the signature, then access the token's data. If you're planning to hand over critical information to the client, signing or encrypting it is the only way forward.

ℹ Are JWTs safe?

JWTs have been under a lot of scrutiny in recent years, partly because of some design flaws that had to be course-corrected, such as the support of a 'None' algorithm, which would effectively allow forging tokens without any prior knowledge of secrets and keys used to sign them. Luciano Mammino, a researcher from Italy, even managed to publish a JWT cracker to illustrate how easy it could be to crack JWTs through brute-forcing, granted the algorithm and secrets used are weak.

In all honesty, JWTs are very useful when you want to exchange data between two parties. For example, you could send a client the URL `https://example.com/check-this-message?token=$JWT` so that they could access the data within the token and know it comes from a trusted source. As session IDs, oftentimes there are simpler mechanisms you could rely on, as you only really need to issue a cryptographically random ID that identifies a client

really need to issue a cryptographically random ID that identifies a client.

Does this mean JWTs are not safe? Not really, as it depends on how you use them. Google, for example, allows authentication to their APIs through JWTs. The trick is to use safe, long secrets or a cryptographically secure signing algorithm, and understand the use-case you're presented with. JWTs also don't make any effort to encrypt the data they hold, and they're only concerned with validating its authenticity. Understand these trade-offs and make your own educated choice.

In addition, you might want to consider PASETO, "Platform Agnostic SEcurity TOkens." They were designed with the explicit goal to provide the flexibility and feature-set of JWTs without some of the design flaws that have been highlighted earlier on.

Further readings:

- paragonie.com/blog/2017/03/jwt-json-web-tokens-is-bad-standard-that-everyone-should-avoid
- kevin.burke.dev/kevin/things-to-use-instead-of-jwt
- www.pingidentity.com/en/company/blog/posts/2019/jwt-security-nobody-talks-about.html

---

In the next lesson, we'll see how session IDs can be generated.