Sjoerd Langkemper

Web application security

Attacking JWT authentication

Sep 28, 2016

JSON Web Tokens or JWTs are used by some web applications instead of traditional session cookies. Because of their statelessness and the signature implementation there are some security issues that are specific to JWTs. This post describes some ways you can verify that a JWT implementation is secure.

About JWTs

What is a JWT

A JWT (JSON Web Token) is a string that contains a signed data structure, typically used to authenticate users. The JWT contains a cryptographic signature, for example a HMAC over the data. Because of this, only the server can create and modify tokens. This means the server can safely put `userid=123` in the token and hand the token to the client, without having to worry that the client changes his user identifier. This way, authentication can be stateless: the server does not have to remember anything about the tokens or the users because all information is contained within the token.

How to recognize a JWT

A JWT typically looks like this:

```
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJpc3MiOiJodHRwOlwvXC9kZW1
```

There are several properties that can help recognizing a JWT:

- JWTs are long, at least 100 characters.
- JWTs consist of base64-encoded data: letters, digits, _ and -.
- JWTs consist of three parts, separated by dots.
- They typically occur in the `Authorization` header with the `Bearer` keyword.

JWTs can be used in any context, but are often used in a header like this:

```
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJh...
```

Attacking JWTs

If you have an application that uses JWT for authentication, there are several things you can try to test the security of the authentication layer.

Check for sensitive data in the JWT

JWTs may look like garbage to the naked eye, but actually they are just base64-encoded data. They are easily decoded, for example by using the website JWT.io. There may be sensitive information stored in the JWT, that is easily discovered this way.

Change the signing algorithm

JWTs are signed (or they should be) to prevent users from changing the data within. There are several algorithms that can be used for signing, for example using a HMAC or using RSA signing. The JWT header contains the algorithm used to sign the JWT, and one flaw of some algorithms is that they trust this JWT header, even though it can be manipulated by the client. If this vulnerability is present, the client can create its own tokens, something that the signature is meant to prevent.

Changing the algorithm to none

A JWT header looks like this:

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

This data is base64 encoded and is the part before the first dot of any JWT. The `alg` field here indicates the algorithm used to sign the JWT. One special "algorithm" that all JWT libraries should support is `none`, for no signature at all. If we specify `none` as algorithm in the header and leave out the signature, some implementations may accept our JWT as correctly signed.

Try it

Go to this HS256 demo page that provides a HS256 signed token. Decode the header and change the algorithm to `none`. Remove the signature, but leave the last dot. Send it to the demo page and check whether the token is accepted.

Changing the algorithm from RS256 to HS256

The algorithm HS256 uses a secret key to sign and verify each message. The algorithm RS256 uses a private key to sign messages, and a public key to verify them. If we change the algorithm from RS256 to HS256, the signature is now verified using the HS256 algorithm using the public key as secret key. Since the public key is not secret at all, we can correctly sign such messages.

Consider the following example code, which could be present at the server:

```
jwt = JWT.decode(token, public_key)
```

If the JWT uses asymmetric RS256, this correctly verifies the signature on the token. If the JWT uses symmetric HS256, however, the signature is compared to a HMAC of the token, where the `public_key` is used as key. We can thus exploit this vulnerability by signing our own token using HS256 with the public key of the RS256 algorithm.

Try it

Go to this RS256 demo page. You get a RS256 signed token. Create a new token, set the algorithm to HS256 and sign it with the public key. Verify that the key is accepted.

Read more

You can read more about this attack here:

- JWT Attack Walk-Through
- JWT: Signature-vs-MAC attacks

Crack the key

As previously stated, the HS256 algorithm uses a secret key to sign and verify messages. If we know this key, we can create our own signed messages. If the key is not sufficiently strong it may be possible to break it using a brute-force or dictionary attack. By trying a lot of keys on a JWT and checking whether the signature is valid we can discover the secret key. This can be done offline, without any requests to the server, once we have obtained a JWT.

There are several tools that can brute force the HS256 signature on a JWT:

- jwtbrute, a .NET implementation.
- This Python script I wrote that uses PyJWT to do the decoding.
- John the Ripper

Using John

To use John the Ripper you need a recent version that supports the JWT format. You probably need to compile the latest version from source to get JWT support. This works like this:

```
git clone https://github.com/magnumripper/JohnTheRipper
cd JohnTheRipper/src
./configure
make -s clean && make -sj4
cd ../run
./john jwt.txt
```

John has a size limit on the data it will take. If you run into this limit, consider changing `SALT_LIMBS` in the source code.

Try it

Obtain a JWT from the HS256 demo page. Use one of the above tools to crack the secret. Then, create your own token and sign it with the discovered secret key.

Conclusion

As with many things JWT is fundamentally secure, but some implementations are not. The application may store sensitive information in the JWT, allow changing the signing algorithm, or have a insufficiently strong key used in the signature.

The source for the demo pages can be found on GitHub.

Sjoerd Langkemper works as a penetration tester on web applications in Haarlem, The Netherlands.

Sjoerd Langkemper

Web application security
sjoerd-2019@linuxonly.nl