JSON Web Tokens, bypass authentications throught the NONE hashing algorithm problem

...
Department and University Name

Objectives

This poster shows a quick overview of JWTs with a particular focus on:

- What is a JWT
- Where JWTs are often used.
- How to read JWTs.
- Exploit JWTs most common misconfiguration.

Intro to JWT

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 the HMAC algorithm) or a public/private key pair using RSA. Some of the advantages of JWTs are:

- Compact: Because of their smaller size, JWTs can be sent through a URL, POST parameter, or inside an HTTP header. Additionally, the smaller size means transmission is fast.
- Self-contained: The payload contains all the required information about the user, avoiding the need to query the database more than once.

Figure : Figure caption

Use Cases

Here are some scenarios where JSON Web Tokens are useful:

- Authentication: This is the most common scenario for using JWT. Once the user is logged in, each subsequent request will include the JWT, allowing the user to access routes, services, and resources that are permitted with that token. Single Sign On is a feature that widely uses JWT nowadays, because of its small overhead and its ability to be easily used across different domains.
- Information Exchange: JSON Web Tokens are a good way of securely transmitting information between parties, because as they can be signed, for example using public/private key pairs, you can be sure that the senders are who they say they are. Additionally, as the signature is calculated using the header and the payload, you can also verify that the content hasn't been tampered with.

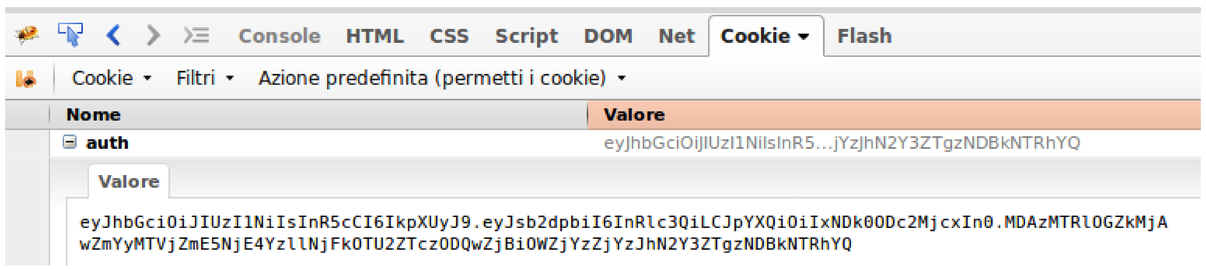

Figure : JWT used as cookie and passed as authentication token

JWTs Structure

JSON Web Tokens consist of three parts separated by dots (.), which are a Header, a Payload and the Signature, all encoded as Base64 values. In particular, a JWT typically looks like the following:

Base64(Header).Base64(Data).Base64(Signature)

The header contains informations about the hashing algorithm used to encode the signature:

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

JWTs Sctructure (cont.)

The payload contains the actual data in transit, for example in this case, the username of the logged user, that can be used by the server to know if it is authorized to view the current page:

```
{
  "sub": "1234567890",
  "user": "notAdmin",
}
```

Finally the signature is composed by the combination of the header and the payload and encoded trhought the chosen algorithm:

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

Vulnerability

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. Interestingly enough, it is one of only two algorithms that are mandatory to implement (the other being HS256).
Unfortunately, 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.
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. This is a good idea, but it doesn't solve the underlying problem: attackers control the choice of algorithm, and in any case the signature can always be left blank.

Exploit

To exploit this vulnerability, you just need to decode the JWT and change the algorithm used for the signature. Then you can submit your new JWT. However, this won't work unless you remove the signature. If you think like a developer, it actually makes a lot of sense. If you had to develop a JWT library, you will start by generating all the signatures you need to support as it allows you to compare your implementation to others. Then you will create a generic verification method that compare what you get from the JWT with the signature you are generating (based on the algorithm in the header). Since you don't generate any signature with the None algorithm, you will ensure that the signature is empty. Therefore, as an attacker, you need to provide an empty signature.

References

Acknowledgements

Nam mollis tristique neque eu luctus. Suspendisse rutrum congue nisi sed convallis. Aenean id neque dolor. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas.

Contact Information