JWT ATTACK

What are JWT Token?

JSON web tokens (JWTs) are a standardized format for sending cryptographically signed JSON data between systems. They can theoretically contain any kind of data but are most used to send information ("claims") about users as part of authentication, session handling, and access control mechanisms.

Unlike with classic session tokens, all of the data that a server needs is stored client-side within the JWT itself. This makes JWTs a popular choice for highly distributed websites where users need to interact seamlessly with multiple back-end servers.

What is the structure of a JWT?

Header

A header in a JWT is mostly used to describe the cryptographic operations applied to the JWT like signing/decryption technique used on it. It can also contain the data about the media/content type of the information we are sending.This information is present as a JSON object then this JSON object is encoded to BASE64URL. The cryptographic operations in the header define whether the JWT is signed/unsigned or encrypted and are so then what algorithm techniques to use. A simple header of a JWT looks like the code below:

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

The ‘alg’ and ‘typ’ are object key’s having different values and different functions like the ‘typ’ gives us the type of the header this information packet is, whereas the ‘alg’ tells us about the encryption algorithm used.

Note: HS256 and RS256 are the two main algorithms we make use of in the header section of a JWT.

Some JWT's can also be created without a signature or encryption. Such a token is referred to as unsecured and its header should have the value of the alg object key assigned to as 'none'.

```
{
  "alg":"none"
}
```

Payload

The payload is the part of the JWT where all the user data is added. This data is also referred to as the 'claims' of the JWT. This information is readable by anyone so it is always advised to not put any confidential information in here. This part generally contains user information. This information is present as a JSON object then this JSON object is encoded to BASE64URL. We can put as many claims as we want inside a payload, though unlike header, no claims are mandatory in a payload. The JWT with the payload will look something like this:

```
{
   "userId":"b07f85be-45da",
   "iss": "https://provider.domain.com/",
   "sub": "auth/some-hash-here",
   "exp": 153452683
}
```

The above JWT contains userId,iss,sub,and exp. All these play a different role as userId is the ID of the user we are storing, 'iss' tells us about the issuer, 'sub' stands for subject, and 'exp' stands for expiration date.

Serialized

JWT in the serialized form represents a string of the following format:

[header].[payload].[signature]

all these three components make up the serialized JWT. We already know what header and payload are and what they are used for.Let’s talk about signature.

Signature

This is the third part of JWT and used to verify the authenticity of token. BASE64URL encoded header and payload are joined together with dot(.) and it is then hashed using the hashing algorithm defined in a header with a secret key. This signature is then appended to header and payload using dot(.) which forms our actual token header.payload.signature

Syntax:

HASHINGALGO(base64UrlEncode(header) + “.” + base64UrlEncode(payload),secret)

So, all these above components together are what makes up a JWT. Now let’s see how our actual token will look like:

JWT Example:

header:

```
{
 "alg": "HS256",


 "typ": "JWT"
}
```

Payload:

{

 "id" : 123456789,

 "name" : "Joseph"

}

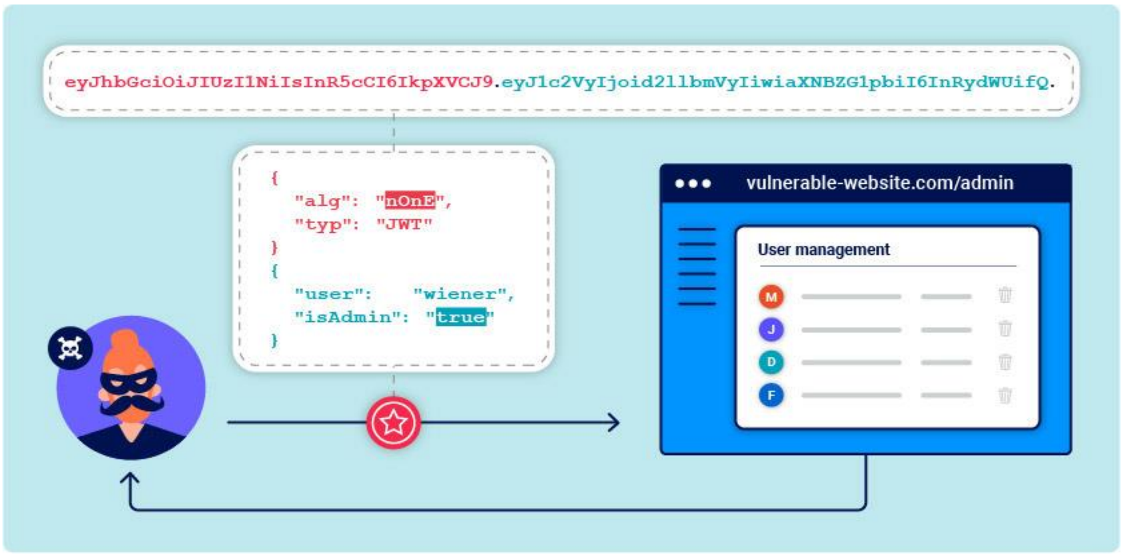
Secret: GeeksForGeeks

JSON Web Token

EyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTIzNDU2Nzg5LCJuYW1lIjoiSm9zZXBoIn0.OpOSSw7e485LOP5PrzScxHb7SR6sAOMRckfFwi4rp7o

What is JWT Token Hacking and the types of attacks?

When implemented correctly, JSON web tokens provide a secure way to identify the user since the data contained in the payload section cannot be tampered with. (Since the user does not have access to the secret key, she cannot sign the token herself.)



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjoid2llbmVyIiwiaXNBZG1pbiI6InRydWUifQ.
```

```
{
  "alg": "nOnE",
  "typ": "JWT"
}
{
  "user":    "wiener",
  "isAdmin": "true"
}
```

But if implemented incorrectly, there are ways that an attacker can bypass the security mechanism and forge arbitrary tokens.

Change up the algorithm type

One of the ways that attackers can forge their own tokens is by tampering with the **alg** field of the header. If the application does not restrict the algorithm type used in the JWT, an attacker can specify which algorithm to use, which could compromise the security of the token.

1. None algorithm

JWT supports a **“none”** algorithm. If the **alg** field is set to “none”, any token would be considered valid if their signature section is set to empty. For example, the following token would be considered valid:

eyAiYWxnIiA6ICJOb25lIiwgInR5cCIgOiAiSldUIiB9Cg.eyB1c2VyX25hbWUgOiBhZG1pbiB9Cg.

It is simply the base64url encoded versions of these two blobs, and no signature is present.

{

"alg": "none",

"typ": "JWT"

}{

"user": "admin"

}

This feature was originally used for debugging purposes. But if not turned off in a production environment, it would allow attackers to forge any token they want by setting the **alg** field to “none”. They can then impersonate anyone on the site by using the forged tokens.

2. HMAC algorithm

The two most common types of algorithms used for JWTs are HMAC and RSA. With HMAC, the token would be signed with a key, then later verified with the same key. As for RSA, the token would first be created with a private key, then verified with the corresponding public key.

HMAC -> signed with a key, verified with the same keyRSA -> signed with a private key, verified with the corresponding public key

It is critical that the secret key for HMAC tokens and the private key for RSA tokens are kept a secret since they are used to sign the tokens.

Now let’s say that there is an application that was originally designed to use RSA tokens. The tokens are signed with a private key A, which is kept a secret from the public. Then the tokens are verified with public key B, which is available to anyone. This is okay as long as the tokens are always treated as RSA tokens.

Token signed with key A -> Token verified with key B (RSA scenario)

Now if the attacker changes the **alg** to HMAC, she might be able to create valid tokens by signing the forged tokens with the RSA public key B.

This is because originally when the token is signed with RSA, the program verifies it with the RSA public key B. When the signing algorithm is switched to HMAC, the token is still verified with the RSA public key B, but this time, the token can be signed with the same public key B (since it’s using HMAC).

Token signed with key B -> Token verified with key B (HMAC scenario)

Provide a non-valid signature

It is also possible that the signature of the token is never verified after it arrives at the application. This way an attacker could simply bypass the security mechanism by providing an invalid signature.

Bruteforce the secret key

It could also be possible to brute force the key used to sign a JWT.

The attacker has a lot of information to start with: she knows the algorithm used to sign the token, the payload that was signed and the resulting signature. If the key used to sign the token is not complex enough, she might be able to brute force it easily.

Leak the secret key

If an attacker is not able to brute force the key, she might try leaking the secret key instead. If another vulnerability (like a directory traversal, XXE, SSRF) exists that allows the attacker to read the file where the key value is stored, the attacker can steal the key and sign arbitrary tokens.

KID manipulation

KID stands for “Key ID”. It is an optional header field in JWTs, and it allows developers to specify the key to be used for verifying the token. The proper usage of a KID parameter looks like this:

```
{
 "alg": "HS256",
 "typ": "JWT",
 "kid": "1"      // use key number 1 to verify the token
}
```

Since this field is controlled by the user, it can be manipulated by attackers and lead

1.Directory traversal

Since the KID is often used to retrieve a **key file** from the file system, if it is not sanitized before use, it can lead to a directory traversal attack. When this is the case, the attacker would be able to specify any file in the file system as the key to be used to verify the token.

“kid”: “../../public/css/main.css” // use the publicly available file main.css to verify the token

For example, the attacker can force the application into using a publicly available file as the key, and sign an HMAC token using that file.

2. SQL injection

The KID could also be used to retrieve the key from a **database**. In this case, it might be possible to utilize SQL injection to bypass JWT signing.

If SQL injection is possible on the KID parameter, the attacker can use this injection to return any value she wants.

“kid”: "aaaaaaa' UNION SELECT 'key';--"// use the string "key" to verify the token

For example, this above injection will cause the application to return the string “key” (since the key named “aaaaaaa” doesn’t exist in the database). The token will then be verified with the string “key” as the secret key.

Header parameter manipulation

In addition to a key ID, JSON web token standards also provide developers with the ability to specify keys via a URL.

JKU header parameter

JKU stands for “JWK Set URL”. It is an optional header field used to specify a URL that points to a set of keys that are used to verify the token. If this field is allowed and not properly restricted, an attacker could host their own key file and specify that the application uses it to verify tokens.

jku URL -> file containing JWK set -> JWK used to verify the token

2. JWK header parameter

The optional **JWK** (JSON Web Key) header parameter allows attackers to embed the key used to verify the token directly in the token.

3. X5U, X5C URL manipulation

Similar to the jku and jwk headers, the **x5u and x5c** header parameters allow attackers to specify the **public key certificate or certificate chain** used to verify the token. x5u specifies the information in the URI form while x5c allows the certificate values to be embedded in the token.

Other JWT security issues

There are also other JWT issues that arise when they are not correctly implemented. These are not very common, but definitely worth looking out for:

Information leak

Since JSON web tokens are used for access control, they often contain information about the user.

If the token is not encrypted, anyone can base64 decode the token and read the token's payload. So if the token contains sensitive information, it might become a source of information leaks. A properly implemented signature section of the JSON web token provides data integrity, not confidentiality.

Command injection

Sometimes when the KID parameter is passed directly into an insecure file read operation, it is possible to inject commands into the code flow.

One of the functions that could allow this type of attack is the **Ruby open()** function. This function allows attackers to execute system commands by simply tacking the command onto the input after the KID file name:

"key_file" | whoami;

This is just one example. Theoretically, vulnerabilities like this can happen whenever an application passes any of the header parameters unsanitized into any function resembling system(), exec(), etc.

Ultimately, JSON web tokens are just another form of user input. They should always be handled with skepticism and sanitized rigorously.

Impact

· Sensitive Information Disclosure

· The authenticity of client can be compromised

· It leads Account takeover

· Attackers can access the Server Files

· they might be able to read data from the underlying SQL database

Mitigations

To prevent such attacks, applications typically use URL filtering. Unfortunately, there are ways for attackers to bypass such filtering, including:

- Using https://trusted (for example https://trusted@attacker.com/key.json), if the application checks for URLs starting with trusted
- Using URL fragments with the # character
- Using the DNS naming hierarchy
- Chaining with an open redirect
- Chaining with a header Injection
- Chaining with SSRF

For this reason, it is very important for the application to whitelist permitted hosts and have correct URL filtering in place. Beyond that, the application must not have other vulnerabilities that an attacker might chain to bypass URL filtering.

REFERENCE

- JWT attacks | Web Security Academy (portswigger.net)
- JSON Web Token attacks and vulnerabilities | Invicti