REDHUNT LABS

DISCOVER. ATTACK. REPEAT.

A Practical Guide to Attacking JWT (JSON Web Tokens)

Technological advancements, digitalization, and increased use of technology have resulted in a significant evolution. As the complexity of web applications increased, their dependencies on other micro-services increased. As a result, in the new micro-service architecture, JSON Web Tokens had the upper hand over traditional stateful session-based authentication methods. JWTs are quickly becoming the preferred format for securely exchanging data between clients and intermediaries as they have lower latency for authentication, don't need a centralized database, are stateless in nature, and provide exemplary implementations to prevent nefarious activities.

This guide will provide a comprehensive overview of JSON Web tokens, how it works, and the various JWT-related attacks.

So let’s start, shall we?

TABLE OF CONTENTS

01 What are JSON Web Tokens?

02 Structure of JWT tokens

03 How does JWT actually work?

04 Signing Algorithms

05 Types of tokens

06 JWT Attack Scenarios

07 Other Issues

08 JWT Best Practices

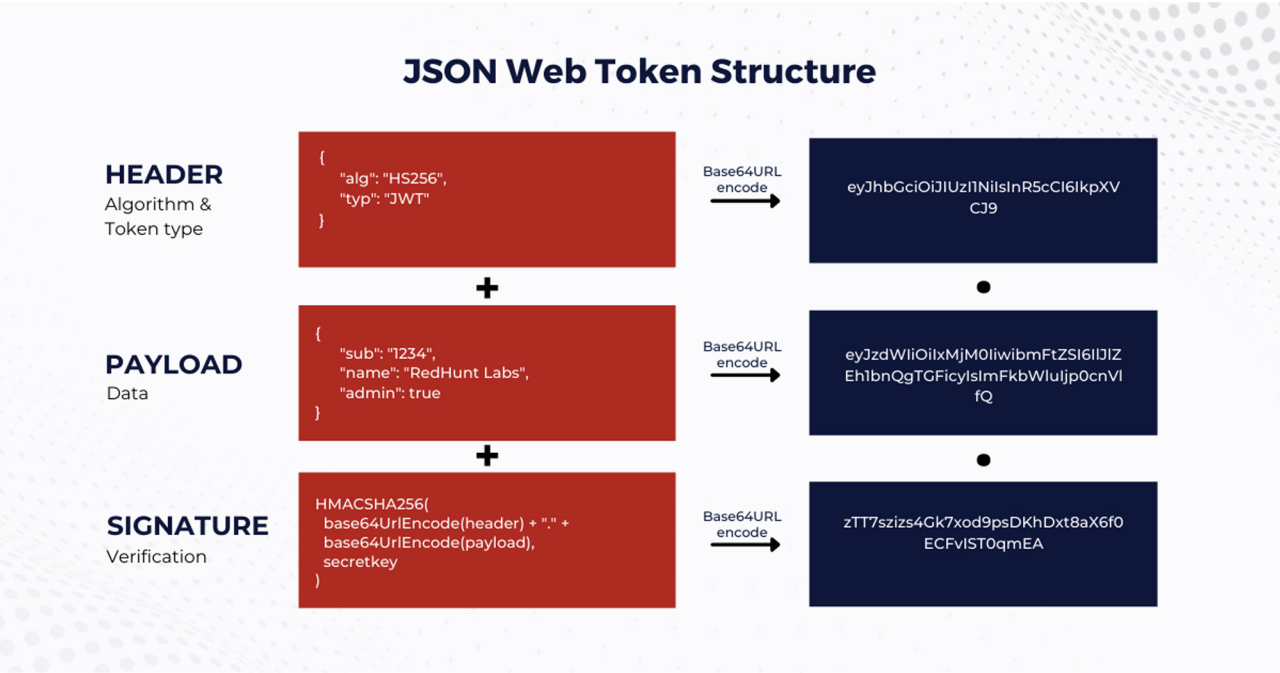
What are JSON Web Tokens ?

JSON Web Token (JWT) is a standard for creating tokens that assert a set of claims. These claims represent the user's identity and thus define the activities they are authorized to perform. JWT's contain these claims as JavaScript Object Notation(JSON) objects, which are digitally signed by the issuer of the token. Following successful authentication, the user is assigned a JSON Web Token, which serves as a user identity badge and must be sent with each subsequent request.

JWT's are a secure way of transmitting information between parties. Since a JWT contains all the required information, the application doesn't need to query the database more than once.

Structure of JWT tokens

A JWT consists of 3 Base64-URL encoded sections concatenated by a period(.); namely:- **the Header**, **the payload**, and **the Signature**. Let's go through each one of them:



The Header

The header is typically the first section of the JWT, which contains the metadata about the token. Typically, it identifies the token type("**typ**") and the hashing algorithm("**alg**") used encoded in JSON format.

The Payload

The next section of the JWT is the payload section that contains the claims. These claims provide identifying information about the

logged-in user. A claim can be observed as statements about an entity, additional data such as description, which the server will verify. There are no restrictions on the payload's size; however, we should aim to keep it as short as possible.

To maintain interoperability among various services, some standards have been set in place to define what and how certain data is communicated. There are three types of claims defined by JWT:

Registered Claims:

Certain pre-defined claims are registered in the IANA JSON Web Token Claims registry. While not intended to be obligatory, these JWT claims can serve as an initial guide to creating effective, compatible claims.

Public claims:

These are the custom claims that we can define using all the alphanumeric characters. The names of public claims must be collision-resistant. We can register them with the IANA JSON Web Tokens Claims registry or use a collision-resistant name to prevent a collision. For example: a Universally Unique Identifier (UUID), or an Object Identifier (OID).

Private Claims:

These claims are a set of custom claims which are more specific to the organization or the application. Private claim names may be

utilized inside a company where JWTs are only transferred between known systems. In addition, user IDs, user roles, and any other information can be defined by us. These are subject to collision, so use them with caution.

3) The Signature

The third and last section of the JWT token is the signature part. The JWT standard follows the JSON Web Signature (JWS) specification to generate the final signed token. First, the signature is calculated by first Bas64-URL encoding the header and payload section and concatenating them with a period(.) separator. This output is then fed to an encryption algorithm such as HAMC or RSA to create the JWT Signature section.

This how we get generate the signature:

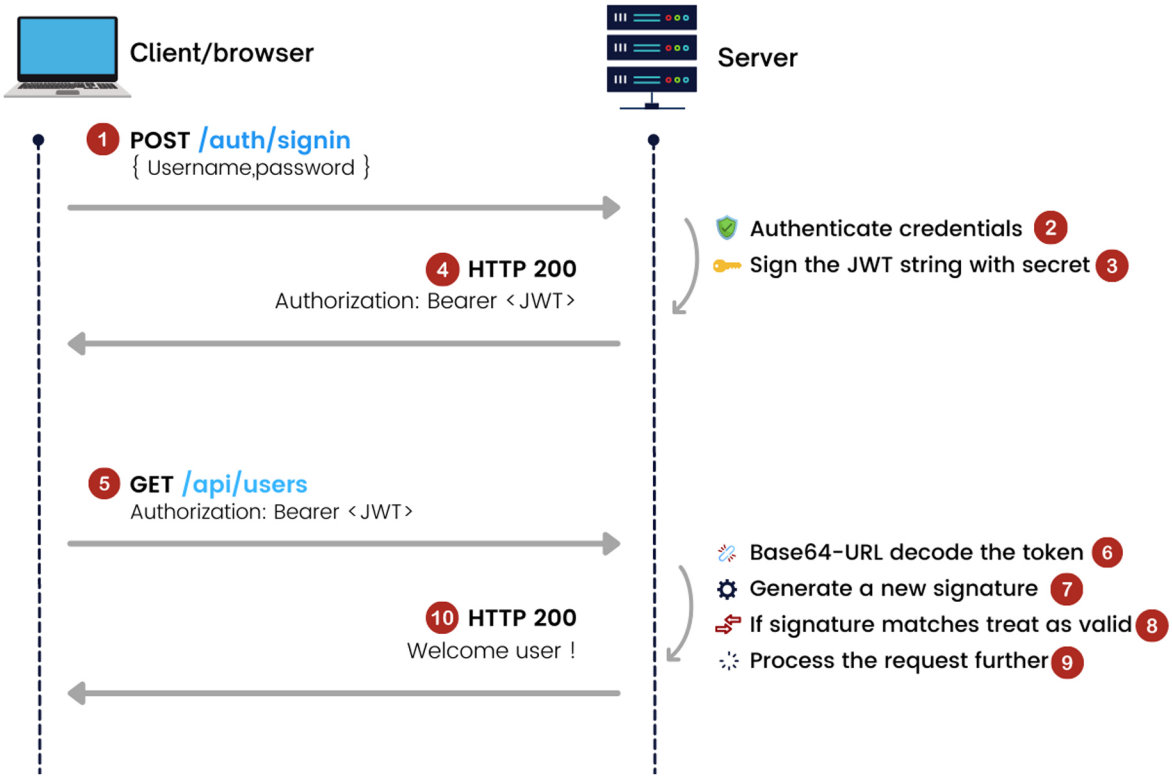
```
1 var encodedString = base64UrlEncode(header) + '.' + base64UrlEncode(header)
2 HMACSHA256(encodedString, 'secret')
```

Although it's crucial to note that a JWT is not encrypted; as a result, any sensitive information is inserted in the token will be viewable by anyone who intercepts it. Hence, we must avoid including sensitive information in the payload portion.

How does JWT actually work ?

JSON Web Tokens are commonly used for token-based authentication. After a user successfully authenticates and authorizes access, all the subsequent requests contain an access token, which the application uses as a credential when calling the target API. The passed token informs the API that the bearer of the token has been granted permission to access the API and perform the actions specified by the scope granted during authorization.

Let's take a broader look at this,



This process can be classified into 2 parts:

a) Obtaining a JWT token

1. First, the user submits a request with login credentials to the authentication server.
2. The authentication server validates the user credentials and creates a JWT token with a payload section containing information that uniquely identifies the user.
3. The authentication server then takes the secret key and uses it to sign the header and the payload section. This constitutes a complete JSON Web Token.
4. The token is sent back to the user, typically in the Authorization header using the Bearer schema.
   The content of the header should look like the following:
   Authorization: Bearer <token>
5. All the subsequent requests made to the server from now onwards must include token this JWT token. The JWT now acts as temporary user credentials.

b) Accessing resources using the JWT token

6. When an application/resource server receives a request containing a JWT, the JWT token is Base64-URL decoded first.
7. Now it retrieves the algorithm type from the header section. Using the received header and payload section, now it generates its own hash and encrypts it's using the secret key.
8. If the new signature matches with the signature received then the server treats the JWT as a valid one.
9. Further, since the payload part includes user identifiable information, we can now make authorized access to the required resources.

Signing Algorithms :

The purpose of the signature section is to allow intermediaries to verify the authenticity of the JWT token, ensuring that it has not been tampered with. The process of checking the signature of a JWT is known as validation or validating a token. Various signing algorithms are used to sign the header and payload section. These algorithms are broadly divided into two types:

a) Symmetric key algorithms:

In a symmetric-key algorithm, a single secret key is used to encrypt and decrypt the data. One of the commonly used symmetric key algorithms is the HMAC.

HMAC

Hash-based Message Authentication Code (HMAC) takes a hash function, a message, and a secret key as inputs and produces the output. The cryptographic hash function's strength ensures that the message cannot be altered without the secret key. Using a weak hash function may allow malicious users to compromise the validity of the output. Hence strong hash functions must be used along with HMAC's.

HMAC + SHA256 (HS256)

The SHA2 family of hash functions is still strong enough for today's standards. The hashes of the header and payload sections are generated using the SHA256 hashing algorithm. These signing algorithms facilitate the easy creation and validation of tokens and should be used when all intermediary parties can be trusted to secure our secret key.

b) Asymmetric key algorithms:

In asymmetric key algorithms, 2 different keys are used to encrypt and decrypt the data. The private key is a secret is used to encrypt the data, while the public key being publicly available is used to decrypt the data.

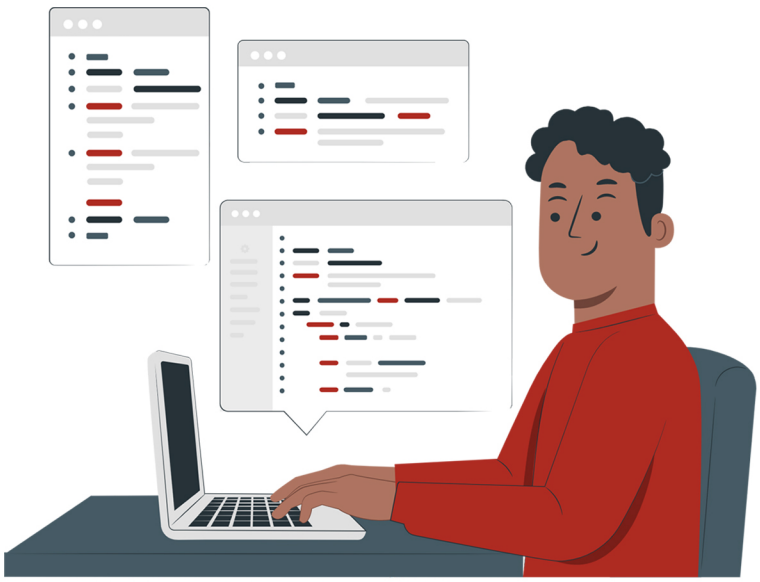
RSA + SHA256 (RS256)

The identity provider has the private key, which is used to generate the signature, while the JWT token consumer gets the public key to verify the signature. RSA algorithms are commonly used in micro-service architectures where you cannot trust the opposite party by distributing your private key. Doing so will give the party the power to generate arbitrary tokens, and the party can in turn, get the capacity to create arbitrary tokens on their own. Hence, this would be a severe threat.

There are several other algorithms available that can be used to create a JWT:

Algorithm	Specification
HS384	HMAC USING SHA-384
HS512	HMAC USING SHA-512
RS384	RSASSA-PKCS1-V1_5 USING SHA-384
RS512	RSASSA-PKCS1-V1_5 USING SHA-512
ES256	ECDSA USING P-256 AND SHA-256
ES384	ECDSA USING P-384 AND SHA-384
ES512	ECDSA USING P-512 AND SHA-512
PS256	RSASSA-PSS USING SHA-256 AND MGF1 WITH SHA-256
PS384	RSASSA-PSS USING SHA-384 AND MGF1 WITH SHA-384
PS512	RSASSA-PSS USING SHA-512 AND MGF1 WITH SHA-512

https://owasp.org/www-chapter-vancouver/assets/presentations/2020-01_Attacking_and_Securing_JWT.pdf



Types of tokens :

There are multiple sorts of tokens, but the most common in JWT authentication are access tokens and refresh tokens.

a) Access Tokens:

When a user authenticates, the user is assigned an access token to make authorized calls to the API server. It contains all of the information the server requires to determine whether or not the user or device can access the resource you are requesting. However, these access tokens have a short lifespan period after which the user will no longer be allowed to make authorized requests to the application server. The issuer party can control the lifespan of an access token.

These access tokens contain claims like iat: meaning “Issued at”, exp: meaning “Expiration Time” which give us an idea about when they will expire.

```
{
  "alg": "HS256",
  "typ": "JWT"
}.
{
  "name": "John Doe",
  "role": "User",
  "iat": 1516239022,
  "exp":1516309443
}.SIGNATURE

```

b) Refresh Tokens:

A refresh token is a special kind of token that is used to receive a new access token. If the access token expires, the user would have to re-authenticate themselves. This problem can be skipped if we implement the usage of refresh tokens. As the access token reaches its expiry time, the refresh token makes an API call and obtains a new access token from the server.

Refresh tokens improve the authentication process significantly as now the user has only to authenticate itself once, and subsequent authentication can take place using the refresh tokens.

```
{
  "token": "eyJhb....GciOiJSUR5cCI6IkpXYeIf3T_Hv5p5V2w",
  "refreshToken": "YRjxL....psjRuKcy_9msxg"
}
```

JWT Attack Scenarios :

The majority of JWT attacks are caused by implementation errors rather than flaws in the design of Web Tokens. This section will look at different types of attack scenarios and techniques that can be used to exploit a JWT based application.

a) Failing to verify Signature

A JWT's signature ensures that the header and payload have not been altered by any bad actors. But the concern arises when the application server is not verifying this signature. Sometimes developers fail to implement proper methods that verify the signature with every request on the server-side. As a result, the attacker gains the ability to modify the payload portion, which may result in unauthorized resource access or privilege escalation.

```
{
  "alg": "HS256",
  "typ": "JWT"
}.
{
  "name": "John Doe",
  "is_admin": "true"
}.SIGNATURE
```

b) NONE Algorithm attack

JWT libraries include support for the NONE algorithm. This algorithm is intended to be used for situations where the token's integrity has already been verified. Interestingly this algorithm specifies that the token is not signed and will not contain the signature part. If the application server fails to verify the "**alg**" value in the JWT header or interprets the `NONE` algorithm, indicates that we can submit a JWT without the signature section. Since the signature section is missing, a malicious user can forge a JWT with a modified payload section and happily gain unauthorized access. That is why it is important to not accept tokens with `None, none, NONE, nOnE,` or any other case variations as the "**alg**" value.

```
{
  "alg": "NONE",
  "typ": "JWT"
}.
{
  "name": "John Doe",
  "is_admin": "true"
}.SIGNATURE
```

c) Weak HMAC keys

Often JWT is generated using symmetric signing algorithms such as HS256, which uses the same shared secret key to sign and verify the signature. In such algorithms, if the secret key is too short in length or easily guessable can prove to be dangerous.

Using various automated tools, the attacker can easily crack the secret key. Once the secret key is known, the attacker can generate arbitrary tokens which could prove a threat to the organization. For bruteforcing the secret key we can use a python based called JWT Tool. It can be more critical as it is possible to brute-force the secret locally.

```
python3 jwt_tool.py eyJh....bGciOiH8Xu45DHuwAsog --crack -d jwt-secrets.txt

<snip>...<snip>

[+] mysecretkey is the CORRECT key!
```

d) Algorithm Confusion Attack

JWT is compatible with both symmetric and asymmetric encryption algorithms.HS256 is a symmetric key algorithm that uses the same secret key to sign and verify the token. In contrast, RS256 is an asymmetric key algorithm where it uses the private key to sign the tokens and the public key to verify the tokens.

If an attacker manages to obtain the public key, malicious tokens can be generated by:

1. Changing the algorithm type from RS256 to HS256
2. Making desired changes in the payload part
3. Then signing the token with the public key
4. Returning this JWT to the application

Since there are no checks for the expected algorithm on the server-side, the server would first parse the header and determine that it is an HMAC algorithm token and send the token for verification. However, we have already generated the malicious JWT token with the open public key. Suppose the server decides to compare the signature to verify the token's authenticity. In that case, it will use the same public key to verify, resulting in identical signatures(as in HMAC same key is used to sign and verify the token). The token will be considered a valid one. This way, the attacker can access those unauthorized resources that were once impossible.

```
{
  "alg": "HS256",
  "typ": "JWT"
}.
{
  "name": "John Doe",
  "is_admin": "true"
}.SIGNATURE
```

e) Attacks using the “jku” parameter

The **jku** parameter in the JWT header can be used to indicate the JSON Web Key Set URL. This parameter specifies where to extract the JSON Web Key(JWK), mostly the public key used to validate the signature.

An attack can be carried out in the following ways:

1. The attacker changes the **jku** parameter value to, its own hosted public key.
2. If URL filtering mechanisms are not implemented, the application server then fetches the key from the URL mentioned in the token's header.
3. Since the attacker signed the token with the private key it possesses, the server will recognize the JWT as a valid one upon verification with the public key.

```
{
  "alg": "HS256",
  "typ": "JWT",
  "jku": "http://attacker.com/key.json"
}.
{
  "name": "John Doe",
  "is_admin": "true"
}.SIGNATURE
```

There are a number of ways for attackers to get through the URL filters like:

- Make sure that the URL is trusted
- Using DNS naming hierarchy
- Chaining with an open redirect
- Chaining with SSRF
- Chaining with header injection

Similar to the "jku" parameter, another parameter called "**x5u**" is used to retrieve remotely-stored X.509 public-key certificates. The attacks mentioned above can also be tried out if the application is using "x5u" parameter in the header.

f) Abusing the kid parameter:

Often authorization servers use multiple secret keys for signing tokens. The "kid" element acts as a key identifier that specifies which key to use while verifying the token's signature. It points to a specific key present in the file system or database.

1) Command injection:

Since the attacker has control over the "**kid**" header parameter, the attacker-controlled value can be passed to the "system" function. The attacker could send a command injection payload to the server and perform malicious activities. The attacker could insert a reverse shell payload to obtain RCE or can then retrieve the secret key itself.

```
{
  "alg": "HS256",
  "typ": "JWT",
  "kid": "key1;python -m SimpleHTTPSServer 1337"
}.
{
  "name": "John Doe",
  "is_admin": "true"
}.SIGNATURE

curl http://target.com:1337/key1
```

2) Path traversal

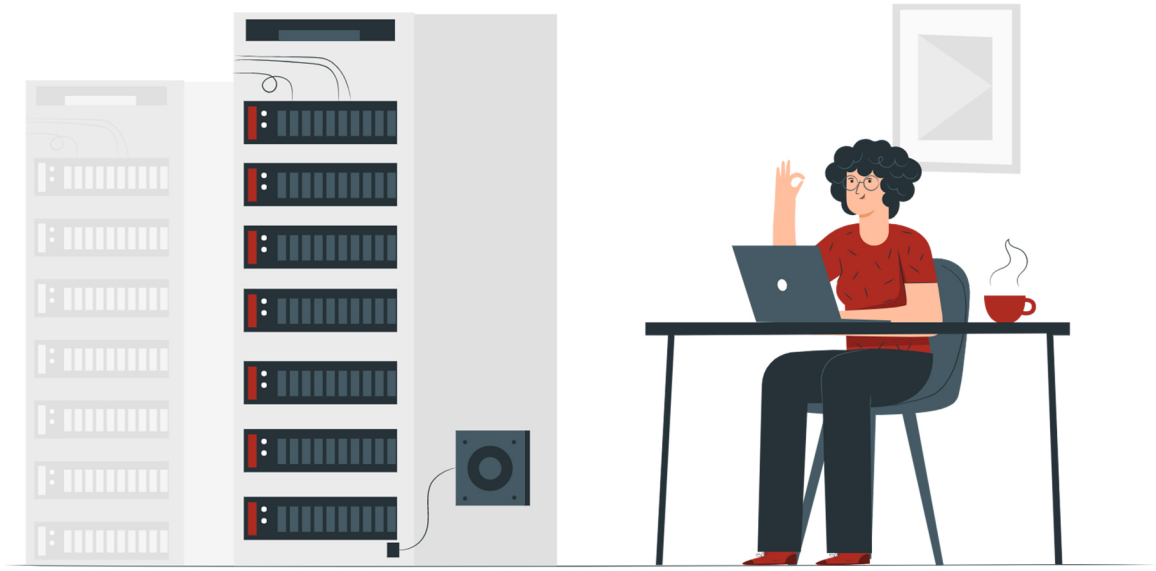
The "**kid**" parameter in the header, specifies the path to the key in the filesystem, which is used to verify the token. Suppose the attacker enters the path to a file with predictable content in the "kid" header parameter. In that case, the attacker can generate a forged token since the secret key is already known. One such file is the `"/proc/sys/kernel/randomize_va_space",` which is used in Linux systems and has predictable values like 0,1,2. The attacker can now create a malicious token using secret values 0,1,2 and send it to the server.

```
{
  "alg": "HS256",
  "typ": "JWT",
  "kid": "../../../../../../../dev/null"
}.
{
  "name": "John Doe",
  "is_admin": "true"
}.SIGNATURE
```

3) SQL Injection

Some applications store their keys in the database. If such a key is referenced in the "**kid**" parameter, it might be vulnerable to SQL injection. In this way, the attacker can perform a SQL Injection attack and can control the value returned to the "**kid**" parameter from the SQL query. Now since the attacker already knows the secret key, the malicious token can be forged.

```
{
  "alg": "HS256",
  "typ": "JWT",
  "kid": "key1' UNION SELECT 'aaa"
}.
{
  "name": "John Doe",
  "is_admin": "true"
}.SIGNATURE
```



Other Issues :

If an attacker cannot forge malicious tokens, they will attempt to steal them. Even if all JWT implementations are perfect, the users may be put at risk if the application is vulnerable to other flaws. These vulnerabilities include:

1) Cross-site scripting(XSS)

Sometimes, the JWT token is stored as a cookie. In such cases, if the application is vulnerable to XSS attack, the payloads can be used to steal them.

2) Cross-Site Request Forgery (CSRF)

When the JWT token is stored in cookies, it will be sent to the server with every authenticated request. If a victim executes a CSRF payload, the JWT token would be sent to the server to induce users to perform actions that they do not intend to perform.

3) Cross-origin resource sharing(CORS) misconfiguration

If an application's CORS policy allows arbitrary origins as well as sending credentials, an attacker can send an XHR request to the server. This way the attacker can steal the token if the application leaks it in the response.

JWT Best Practices :

Below are some of the security practices one must consider while using JWTs'.

1. Use the appropriate algorithms according to your use case.
2. The signing keys should always be kept safe and secure.
3. Select strong signing keys.
4. Validate all the possible claims mentioned in the JWT.
5. Do not add sensitive data in the payload section.
6. Give tokens an expiry.
7. Always use HTTPS connections to transmit tokens.
8. Always perform algorithm verification of the received tokens.
9. In the case of nested tokens, always perform all validations.
10. Do not use JWT as access Tokens.

Find your Exposed Attack Surface.
Take action, and reduce your Attack Surface.

Know more | Free Trial | Request Demo

Discover Attack Repeat

We are a cybersecurity company specializing in Attack Surface Management (ASM) and security consultation to help SMEs and Large Enterprises track and secure their attack surface.
Learn more at: www.redhuntlabs.com

Get the latest cybersecurity research and tips at
www.redhuntlabs.com/blog

REDHUNT LABS
DISCOVER. ATTACK. REPEAT.

For more info follow us