Exploiting JWT vulnerabilities: A complete guide

BY AYOUB · NOVEMBER 7, 2025

Before JSON Web Tokens (JWTs) became popular in today's app development landscape, web applications predominantly used server-side sessions, which presented horizontal scalability issues. JWTs solved this by moving authentication data from the server to the token itself. They are self-contained, stateless and cryptographically signed, checking all the boxes for any use case in application development.

However, when best practices are neglected and standard security specifications are ignored to prioritise other concerns, JWT vulnerabilities can and will arise, often resulting in security flaws that pose a high risk to the affected organisation and its customers.

In this article, we'll explore various methods by which JWTs can be vulnerable to allow for authentication bypasses and injection attacks, demonstrating the importance of testing such implementations and the effectiveness of following best practices.

Let's dive in!

What are JSON web tokens (JWTs)

JSON web tokens are commonly implemented to handle authentication sessions in web services such as web applications, APIs and SPAs, but they also have other common use cases, such as:

- Refresh tokens to obtain a new session token
- Password reset and email confirmation tokens
- Sharing and invitation links
- Other secret tokens that provide (temporary) access to a data object (such as a document)

```
POST /api/auth/login HTTP/2
Host: auth.example.com
Content-Type: application/x-www-form-urlencoded

username=intigriti&password=hunter2
```

```
HTTP/1.1 200 OK
Cache-Control: max-age=86000
Date: Thu, 06 Nov 2025 13:37:37 GMT
Content-Type: application/json

{
  "success": true,
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImludGlncml0aSIsInJvbGUiOiIx
MzM3IiwiaWF0IjoxNzYyNDM2MjU3fQ.M2zyz1bPvlP7DiFAMYk4SB2nB5czBmlhhi70KEwE13s",
  "expiresIn": 3600,
  "user": {
    "username": "intigriti",
    "role": "1337"
  }
}
```

HTTP request of an API that issues JWT after sign in

When the implementation of JWTs is flawed, it can leave the target application vulnerable to several JWT attacks, which we will be discussing later throughout this article. Before we dive into the exploitation of these tokens, let's first deconstruct a standard JSON web token to better help us understand how these issues arise.

Watch our JWT attacks playlist!

If you wish to watch a video instead, feel free to browse our educational playlist dedicated to exploiting JWT vulnerabilities on our YouTube channel.

Deconstructing JSON Web Tokens

By design, JSON Web Tokens consist of 3 parts: the header, payload, and signature. The header includes basic information to instruct the library how to sign and validate the token, the payload contains the claims and other arbitrary data properties that the developer decides to include in the token. Lastly, the signature ensures that both the header and payload cannot be tampered with.

As you can see in the figure below, all 3 parts in a standard JWT token are each separated with a dot (`.`) character and base64 (URLSafe) encoded. Let's dive a bit deeper into each part that constructs a valid JSON Web Token. If you are already familiar with the structure of a typical JWT, feel free to skip ahead to the exploitation part.

Deconstructing a JSON Web Token (JWT)

When the payload part is encrypted, we refer to it as JSON Web Encryption (JWEs).

Header

The header contains metadata acting as instructions for the JWT library to help validate, decrypt and sign the token. Metadata can include the algorithm used, token type, any identifiers (such as `kid`) and other relevant metadata.

In the illustration below, you'll be able to find all relevant metadata properties that can be present in a JWT. Later throughout this article, we will be covering how misconfigurations and a lack of input validation can introduce JWT vulnerabilities.

JSON Web Token (JWT)

JSON Web Token Headers

Property	Purpose	Remarks
alg	Signing algorithm	Required. Can be exploited if "none" accepted or algorithm confusion (RS256→HS256).
typ	Token type identifier	Optional. Usually "JWT" or "at+jwt". Can enable type confusion attacks.
kid	Key identifier for verification	Optional. Possibly vulnerable to injection attacks (path traversal, SQL, command injection).
jku	URL to JWK Set	Optional. Can possibly allow attacker-controlled keys to be specified. Can enable SSRF attacks.
jwk	Embedded public key	Optional. Can possibly allow attacker-controlled keys to be specified.
x5u	URL to X.509 certificate	Optional. Can possibly allow attacker-controlled certificate chains to be specified.
x5c	X.509 certificate chain	Optional. Can possibly allow attacker-controlled certificate chains to be specified. Can enable SSRF attacks.
cty	Content type of payload	Optional. Can enable MIME confusion attacks resulting in possible XXE.

Learn more: blog.intigriti.com/hacking-tools/exploiting-jwt-vulnerabilities

Understanding the JSON Web Token (JWT) header part

Payload

The payload contains the actual JWT claims and any data that the application will need. It is a best practice not to include any sensitive data (such as billing information, emails, or other PII) unless it is encrypted (i.e., a JWE). However, note that submissions regarding missing best practices without it leading to a direct vulnerability are usually rejected as informative findings.

Signature

The signature is required to verify the token's integrity. This means that data cannot be tampered with without the key to sign the token. Later in this article, we will cover how neglecting JWT implementation standards can help introduce JWT vulnerabilities.

We've now covered what JSON Web Tokens are, and the differences between a JSON Web Signature (JWS) and a JSON Web Encryption (JWE). Let's now dive into the JWT exploitation part.

Exploiting JWT vulnerabilities

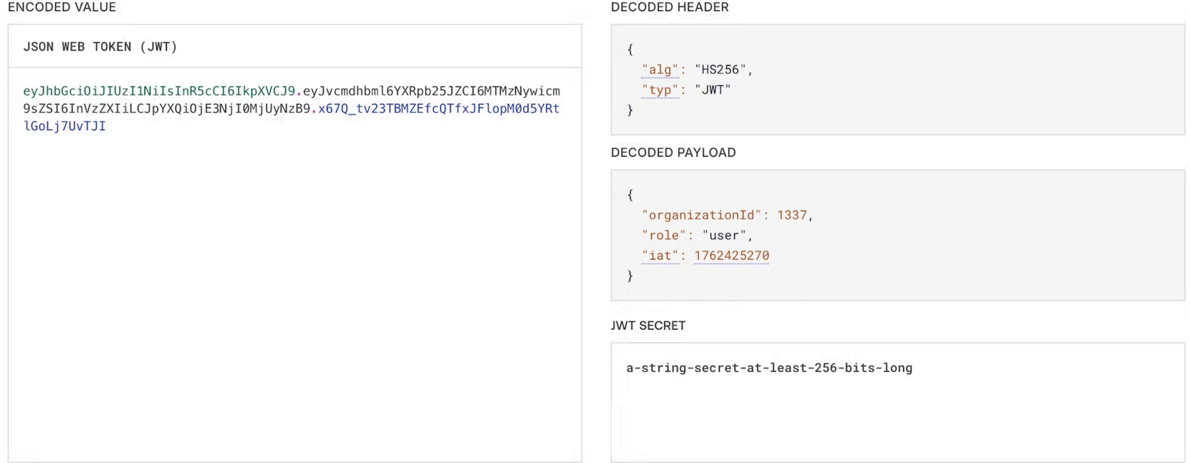
JWT vulnerabilities arise from misconfigurations and improper input validation during the implementation process. Below, we will be covering 7 methods to test for JWT vulnerabilities.

1. None-algorithm allowed

As we saw in the JWT deconstruction section previously in this article, the JWT header contains all metadata to help parse JSON Web Tokens. In some instances, developers enable support for none algorithm, allowing them to issue and use tokens that are not signed, usually for testing purposes or to enable support for unsigned tokens (e.g. non-sensitive sharing links).

However, if incorrect handling is done during production, anyone would be able to tamper with the JWT, including modifying any claims to elevate in-app privileges or exploit injection attacks.

Consider the following JSON web token is used in an accounting platform:



Example of a JWT handling sessions

Any user would be able to:

1. Base64 decode the header and payload
2. Set the **alg** property in the header to **none**
3. Tamper with the claims, in this instance, we'd want to change the **role** property to **owner** and **organizationId** to the victim's organization ID
4. And lastly, omit the signature entirely while leaving the trailing dot

This would ultimately allow us to impersonate any organization owner:

```
GET /api/auth/me HTTP/2
Host: api.example.com
Authorization: Bearer eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJvcmdhbml6YXRpb25JZCI6MTMzOCwicm9sZSI6Im93bmVyIiwiaWF0IjoxNzYyNDI2Njg3fQ.
Content-Type: application/json
```

```
HTTP/2 200 OK
Content-Type: application/json
Content-Length: 96

{
  "success": true,
  "organizationId": 1338,
  “role”: “owner”
}
```

None-algorithm JWT attack

This is not the only instance where developers make mistakes during signature validation. Let's take a look at a few more JWT exploitation examples.

2. Missing signature validation

Similar to the previous misconfigurations, in some cases, developers skip the signature validation altogether when no signature is provided. Often, with the intent to facilitate testing authenticated parts of an application or to provide support for unsigned tokens (e.g. a sharing link for bookmarks or in-app configuration presets).

Once again, when incorrect parsing is performed, it can allow attackers to tamper with the claims and potentially impersonate other users to elevate in-app privileges or exploit injection attacks in case the server incorrectly processes user input.

To test for this case, we'll need to remove the signature altogether, tamper with the JWT claims, and only send the header and payload to the server.

```
GET /api/auth/me HTTP/2
Host: api.example.com
Authorization: Bearer eyJhbGciOiJub25lIiwidHlwIjoiSldUIn0.eyJvcmdhbml6YXRpb25JZCI6
MTMzOCwicm9sZSI6Im93bmVyIiwiaWF0IjoxNzYyNDI2Njg3fQ
Content-Type: application/json
```

```
HTTP/2 200 OK
Content-Type: application/json
Content-Length: 96

{
  "success": true,
  "organizationId": 1338,
  “role”: “owner”
}
```

Missing signature validation JWT attack

As this JWT vulnerability stems from incorrect token parsing, you may need to experiment with the 'alg' header property and pass unintended values like 'none'.

3. JWT algorithm confusion attacks

In case all previous attempts to set the algorithm to 'none' proved to be futile, additional testing is highly recommended, as support for other algorithms may be provided, which can lead to JWT key confusion attacks.

JWT algorithm (or key) confusion attacks originate from developers incorrectly handling the JWT's algorithm, enabling support for other algorithms other than the default used algorithm. This allows an

attacker to specify any other algorithm for the token's signature, such as HS256, which is typically signed with a public key as the secret.

In practice, this usually involves modifying the `alg` property in the JWT's header to `HS256`, tampering with the payload, and signing the token with the server's public key. You'll need to manage to find the exact public key that the server supports, which can usually be found on the server.

4. JWK spoofing

In other scenarios, you'll notice that the JWT parsing library is flawed and allows the inclusion of your own key pair. This may seem unlikely, but [CVE-2018-0114](CVE-2018-0114) is a clear example that originates from a type confusion whereby the attacker can forge tokens by simply including an arbitrary key pair in the token. Let's take a closer look at this CVE and examine how this JWT vulnerability was introduced.

Flawed parsing

The `node-jose` library allowed any unauthenticated attacker to sign tokens as long as the `jwk` property was specified in the header. Back to our table from before, we can see that the `jwk` property has several nested properties that instruct the parsing library to validate the signature:

JSON Web Token (JWT)

JSON Web Token Headers

Property	Purpose	Remarks
alg	Signing algorithm	Required. Can be exploited if "none" accepted or algorithm confusion (RS256→HS256).
typ	Token type identifier	Optional. Usually "JWT" or "at+jwt". Can enable type confusion attacks.
kid	Key identifier for verification	Optional. Possibly vulnerable to injection attacks (path traversal, SQL, command injection).
jku	URL to JWK Set	Optional. Can possibly allow attacker-controlled keys to be specified. Can enable SSRF attacks.
jwk	Embedded public key	Optional. Can possibly allow attacker-controlled keys to be specified.
x5u	URL to X.509 certificate	Optional. Can possibly allow attacker-controlled certificate chains to be specified.
x5c	X.509 certificate chain	Optional. Can possibly allow attacker-controlled certificate chains to be specified. Can enable SSRF attacks.
cty	Content type of payload	Optional. Can enable MIME confusion attacks resulting in possible XXE.

Learn more: blog.intigriti.com/hacking-tools/exploiting-jwt-vulnerabilities

Understanding the JSON Web Token (JWT) header part

Crafting a JSON Web Token with the following header would practically instruct the parsing library to use the attacker's key pair to validate its signature, ultimately, allowing anyone to forge its contents:

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "example-key-id",
  "jwk": {
    "kty": "RSA",
    "kid": "example-key-id",
    "use": "sig",
    "n": "uAPuSn1MG6nFYKjiIcfke-nyMfsZM_Wrea7wlv1l553UrUM8P9VjZ0kTKYX3iyWLDXgyokLsZtqicE5q3c71cQ",
    "e": "AQAB"
  }
}
```

You'll be required to generate your own key pair and transform it into the correct format that the 'n' and 'e' properties need.

This flaw stems from incorrect handling of the **jwk** property, whereby the parsing library trusted any key pair specified by the attacker.

Besides misconfigurations in JWT libraries, there are other oversights, such as injection attacks within a JWT header property and the use of weak secrets. Let's take a look at a few more examples where you can exploit JWT vulnerabilities.

5. Key ID (kid) injection attack

As we've previously seen, misconfigurations can arise when header parameters are incorrectly processed. On some occasions, you may notice that the key ID (kid) is referenced within the header part of the JSON Web Token. This property represents the location of the key file, and essentially instructs the parsing library where to obtain the signing key from on the server's file system.

Some targets employ a set of keys and, therefore, will use the **kid** parameter to identify which key pair was used for the signature. In instances like these, we can test for possible path traversals, SSRFs, and injection attacks, depending on how the parameter is processed. Let's take a look at a few examples.

Exploiting path traversals via JWT kid property

Take the following vulnerable code example into consideration:

```javascript
const jwt = require('jsonwebtoken');
const fs = require('fs');
const path = require('path');

function verifyToken(token) {
  try {
    // Decode the token without verification to get the header
    const decoded = jwt.decode(token, { complete: true });

    if (!decoded || !decoded.header || !decoded.header.kid) {
      return { success: false, payload: null, error: 'Invalid token structure' };
    }

    // Use the 'kid' parameter to locate the key from our trust store
    const keyPath = path.join(__dirname, 'keys', decoded.header.kid);

    // Attempt to read the public key from the specified path
    const publicKey = fs.readFileSync(keyPath, 'utf8');

    // Verify the token with the retrieved public key
    const verified = jwt.verify(token, publicKey);

    return { success: true, payload: verified };
  } catch (error) {
    console.error('Token verification error:', error.message);
    return { success: false, payload: null, error: error.message };
  }
}
```

Path traversal via JWT kid injection

We can clearly see that on **line 15** that the application reads the `kid` parameter and appends it within the file retrieval function without any further validation. In the code example, it points to an unguessable signing key. However, since we can effectively traverse paths, we can make the application fetch the signing key from another directory, as long as it is located on the file system. Our aim is to find a file that returns a guessable key combination, that way, we can replicate the JWT signing process on our machine using the same key.

In practice, exploiting path traversals in JWT kid would look like the following:

1. You modify the claims part according to your needs and the `kid` property in the header part of your JWT to set it to any file located on the server's file system (e.g. `/dev/null`)
2. Sign your forged JWT with the exact same key file (e.g. `/dev/null`)
3. Send your forged JSON Web Token to an endpoint that processes your JWT
4. The vulnerable application will read your forged JWT, locate the key specified in the `kid` property, and finally validate the signature using the retrieved key.

```json
{
  "alg": "RS256",
  "kid": "../../../dev/null"
}
```

This attack essentially allows us to forge JSON Web Tokens and sign them with an arbitrary key as long as it is located on the server's file system. Let's take a closer look at a similar example whereby the signing

key is retrieved from inside a database.

Exploiting SQL injections via JWT kid property

Similar to the previous case, if your target considers the `kid` property whenever it loads the signing key from a database, we could also practically test for SQL, and even NoSQL injections.

Take the following vulnerable code snippet into consideration:

```javascript
const express = require('express');
const jwt = require('jsonwebtoken');
const sqlite3 = require('sqlite3').verbose();
const app = express();

// Connect to database
const db = new sqlite3.Database(...);

function getKeyFromDatabase(keyId, callback) {
  // SQL query to select the key from our trust store
  const query = "SELECT key_data FROM keys WHERE id = '" + keyId + "'";

  db.get(query, (err, row) => {
    if (err) {
      console.error('Database error:', err.message);
      return callback(err, null);
    }

    if (!row) {
      return callback(new Error('Key not found'), null);
    }

    callback(null, row.key_data);
  });
}

// JWT verification middleware
app.use((req, res, next) => {
  const authHeader = req.headers.authorization;

  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    return res.status(401).json({ error: 'Authentication required' });
  }

  const token = authHeader.split(' ')[1];

  try {
    // Decode the token without verification to access the header
    const decodedToken = jwt.decode(token, { complete: true });

    if (!decodedToken || !decodedToken.header || !decodedToken.header.kid) {
      return res.status(400).json({ error: 'Invalid token format' });
    }

    // Retrieve the key using the 'kid' parameter
    getKeyFromDatabase(decodedToken.header.kid, (err, publicKey) => {
      if (err) {
        return res.status(400).json({ error: 'Key retrieval error' });
      }

      try {
        // Verify the token
        req.user = jwt.verify(token, publicKey);
        next();
      } catch (verifyError) {
        res.status(401).json({ error: 'Invalid token' });
      }
    });
  } catch (error) {
    res.status(401).json({ error: 'Authentication failed' });
  }
});
```

SQL Injection via JWT kid injection

On **line 11**, you can see that the `kid` property is concatenated inside an unprepared SQL statement before being evaluated. In this instance, simply breaking out of the context and injecting a predictable payload string would replace the initial signing key with a string value we control.

```
{
    "alg": "RS256",
    "kid": "example-key' OR UNION SELECT 'intigriti'; --"
}
```

Replicating the same proof of concept steps we previously followed would allow us to forge JSON Web Tokens with malicious claims, extract sensitive data from the database, and, under rare conditions, even execute code on the vulnerable server.

6. Bruteforcing common/weak JWT secrets

The use of common or weak secrets can also facilitate forging tokens. With tools like John The Ripper or JWT_tool, we can easily bruteforce weak secrets used to sign and validate the token. Note that this is only possible when the token is signed using a secret. RSA-signed JWSs and JWEs usually require the full key pair, including the private key, which makes guessing attacks almost impossible.

Here's an example of how you'd guess the secret using a single JWT token and a powerful wordlist that includes a possible match:

```
$ john --wordlist=/path/to/wordlist.txt jwt.txt # jwt.txt file contains your JWT token
```

7. Enumerating hard-coded JWT secrets

Secrets are sometimes accidentally pushed to public code repositories, JavaScript files and other configuration files. These can include JWT secrets to sign and validate JSON Web Tokens.

It's always advisable to leverage public information and reconnaissance methodologies such as JavaScript enumeration, GitHub search and Google dorking to spot accidental hard-coded secrets.

Finding a JSON web token secret on the client-side is a strong indication of the JSON web token being signed and validated on the client-side. Make sure you always verify the actual confidentiality of the key.

Conclusion

JSON web token vulnerabilities can arise from various misconfigurations, not following best practices and parsing flaws. In this article, we covered several processing flaws that can occur when developers don't take caution throughout the implementation of JSON web tokens.

So, you've just learned something new about exploiting JWT vulnerabilities... Right now, it's time to put your skills to the test! You can start by practicing on vulnerable labs and CTFs or... browse through our 70+ public bug bounty programs on Intigriti, and who knows, maybe earn a bounty on your next submission!

AUTHOR

Ayoub

Senior security content developer

REQUEST A DEMO

intigriti.com/demo

VISIT THE WEBSITE

intigriti.com

GET IN TOUCH

hello@intigriti.com