Exercises

3.1 *Generate JWT and understand its structure.*

(a) Download the newest Docker image:

```
docker pull docker.io/mazurkatarzyna/jwt-ex1 # if using Docker
podman pull docker.io/mazurkatarzyna/jwt-ex1 # if using Podman
```

If Docker Hub is not responding, use the backup image:

```
docker pull ghcr.io/mazurkatarzynaumcs/jwt-ex1 # if using Docker
podman pull ghcr.io/mazurkatarzynaumcs/jwt-ex1 # if using Podman
```

(b) Using the prepared Docker image, run simple application to generate a JWT:

```
docker run -it --name jwtex1 docker.io/mazurkatarzyna/jwt-ex1
podman run -it --name jwtex1 docker.io/mazurkatarzyna/jwt-ex1
```

If Docker Hub is not responding, use the backup image:

```
docker run -it --name jwtex1 ghcr.io/mazurkatarzynaumcs/jwt-ex1
podman run -it --name jwtex1 ghcr.io/mazurkatarzynaumcs/jwt-ex1
```

Once launched, you will enter the container, which will contain the application template (`app.py` file) using the `PyJWT` library.

(c) Open `app.py` file and edit it with `nano` editor:

```
nano app.py
```

(d) Check out the **docs**, and complete the code with JWT encoding and decoding using the given secret, payload, and algorithm.

(e) What happens if you change the token string before decoding it?

(f) What happens if you use a wrong secret key when decoding?

3.2 *Understand the JWT authorization.*

(a) Download the newest Docker image:

```
docker pull docker.io/mazurkatarzyna/jwt-ex2 # if using Docker
podman pull docker.io/mazurkatarzyna/jwt-ex2 # if using Podman
```

```
docker pull ghcr.io/mazurkatarzynaumcs/jwt-ex2 # if using Docker
podman pull ghcr.io/mazurkatarzynaumcs/jwt-ex2 # if using Podman
```

(b) Using the prepared Docker image, run simple application to test authorization with JWT:

```
docker run -p 5002:5002 --name jwtex2 docker.io/mazurkatarzyna/jwt-ex2
podman run -p 5002:5002 --name jwtex2 docker.io/mazurkatarzyna/jwt-ex2
```

If Docker Hub is not responding, use the backup image:

```
docker run -p 5002:5002 --name jwtex2 ghcr.io/mazurkatarzynaumcs/jwt-ex2
podman run -p 5002:5002 --name jwtex2 ghcr.io/mazurkatarzynaumcs/jwt-ex2
```

Once launched, the application will be available at http://127.0.0.1:5002.
Application has 2 endpoints:

- http://127.0.0.1:5002/login, which allows you to log in (with the following credentials: `username` is `student`, and `password` is `password123`) and returns JWT after successful login,
- http://127.0.0.1:5002/protected, which requires token to access

(c) Use `cURL` to log into the application. Send `POST` request to `/login` endpoint. After successful login, the application should return a JWT.

(d) Use the returned JWT to access `/protected` endpoint. (Simply send `POST` request to `/protected` endpoint with the returned JWT.) What happens if you visit the `/protected` endpoint without providing a JWT?

3.3 *Understand the JWT lifetime.*

(a) Download the newest Docker image:

```
docker pull docker.io/mazurkatarzyna/jwt-ex3 # if using Docker
podman pull docker.io/mazurkatarzyna/jwt-ex3 # if using Podman
```

```
docker pull ghcr.io/mazurkatarzynaumcs/jwt-ex3 # if using Docker
podman pull ghcr.io/mazurkatarzynaumcs/jwt-ex3 # if using Podman
```

(b) Using the prepared Docker image, run simple application to test JWT lifetime:

```
docker run -p 5003:5003 --name jwtex3 docker.io/mazurkatarzyna/jwt-ex3
podman run -p 5003:5003 --name jwtex3 docker.io/mazurkatarzyna/jwt-ex3
```

If Docker Hub is not responding, use the backup image:

```
docker run -p 5003:5003 --name jwtex3 ghcr.io/mazurkatarzynaumcs/jwt-ex3
podman run -p 5003:5003 --name jwtex3 ghcr.io/mazurkatarzynaumcs/jwt-ex3
```

Once launched, the application will be available at http://127.0.0.1:5003.
Application has 2 endpoints:

- http://127.0.0.1:5003/login, which allows you to log in (with the following credentials: `student/password123`) and returns JWT after successful login. (Please note, however, that unlike the previous application, here the JWT token expires after 30 seconds.)
- http://127.0.0.1:5003/protected, which requires token to access

(c) Use `cURL` to log into the application. Send `POST` request to `/login` endpoint. After successful login, the application should return a JWT.

(d) Use the returned JWT to access `/protected` endpoint.

(e) Wait 30 seconds and try to access the `/protected` endpoint again - did you succeed?

Important: JWT tokens include an expiration time to limit how long they remain valid. Once a token is issued and signed, it remains valid until its expiration time. Once a token expires, the server will reject it and the user needs to get a new one. This is a security feature that reduces the risk if a token gets stolen or compromised. Most applications set expiration times anywhere from a few minutes to several hours depending on their security needs. When a token expires, users typically need to refresh it or log in again to continue accessing protected resources.

3.4 *Understand JWT signature verification.*

(a) Using **PyJWT** Python's module, create a JWT token with a simple payload and a secret key. Print the token to see its format (`header.payload.signature`).

(b) Decode the JWT token with the secret key and verification enabled. Print the verified payload.

(c) Decode the JWT token without verifying the signature. Print the decoded payload.

(d) Try modifying the payload and see how decoding behaves with and without verification.

Hint: to solve this task, you can use the following template script:

```
#!/usr/bin/env python3

import jwt
from datetime import datetime, timedelta, timezone
import base64
import json

def b64url_decode(s):
    s += '=' * (-len(s) % 4)
    return base64.urlsafe_b64decode(s)

def b64url_encode(b):
    return base64.urlsafe_b64encode(b).decode().rstrip('=')

SECRET_KEY = "super-secret"
payload = {}

# Decode token without verifying signature
# Decode token WITH signature verification
# Tamper with the payload: decode payload from base64 and create a fake payload
# Re-encode fake payload in base64url
# Build a fake token: header + FAKE payload + original signature
# Try decoding tampered token
```

Template script is also available from the inside of the Docker container:

```
docker run -it --name jwtex4 docker.io/mazurkatarzyna/jwt-ex4
podman run -it --name jwtex4 docker.io/mazurkatarzyna/jwt-ex4
```

If Docker Hub is not responding, use the backup image:

```
docker run -it --name jwtex4 ghcr.io/mazurkatarzynaumcs/jwt-ex4
podman run -it --name jwtex4 ghcr.io/mazurkatarzynaumcs/jwt-ex4
```

Remember to download the newest Docker image:

```
docker pull docker.io/mazurkatarzyna/jwt-ex4 # if using Docker
podman pull docker.io/mazurkatarzyna/jwt-ex4 # if using Podman
```

```
docker pull ghcr.io/mazurkatarzynaumcs/jwt-ex4 # if using Docker
podman pull ghcr.io/mazurkatarzynaumcs/jwt-ex4 # if using Podman
```

3.5 *Understand the JWT signature verification.*

(a) Download the newest Docker image:

```
docker pull docker.io/mazurkatarzyna/jwt-ex5 # if using Docker
podman pull docker.io/mazurkatarzyna/jwt-ex5 # if using Podman
```

(b) Using the prepared Docker image, run simple application to test JWT signature verification:

```
docker run -p 5005:5005 --name jwtex5 docker.io/mazurkatarzyna/jwt-ex5
podman run -p 5005:5005 --name jwtex5 docker.io/mazurkatarzyna/jwt-ex5
```

If Docker Hub is not responding, use the backup image:

```
docker run -p 5005:5005 --name jwtex5 ghcr.io/mazurkatarzynaumcs/jwt-ex5
podman run -p 5005:5005 --name jwtex5 ghcr.io/mazurkatarzynaumcs/jwt-ex5
```

Once launched, the application will be available at http://127.0.0.1:5005. Application has 3 endpoints:

- http://127.0.0.1:5005/token which generates JWT
- http://127.0.0.1:5005/verify which decodes JWT with verifying signature
- http://127.0.0.1:5005/noverify which decodes JWT without verifying signature

(c) Write a Python script that constructs a tampered JWT by changing the payload but keeping the original header and signature of the JWT. The script should send the tampered token to two endpoints on the demo server (http://127.0.0.1:5005/noverify and http://127.0.0.1:5005/verify) and compare the responses.

You can write your code inside a Docker container, just run:

```
docker run -it --network=host docker.io/mazurkatarzyna/jwt-notes
podman run -it --network=host docker.io/mazurkatarzyna/jwt-notes
```

If Docker Hub is not responding, use the backup image:

```
docker run -it --network=host ghcr.io/mazurkatarzynaumcs/jwt-notes
podman run -it --network=host ghcr.io/mazurkatarzynaumcs/jwt-notes
```

Hint: to solve this task, you can use the following template script:

```
#!/usr/bin/env python3

import base64
import json
import requests

SERVER = "http://127.0.0.1:5001"

def b64url_decode(s):
    s += '=' * (-len(s) % 4)
    return base64.urlsafe_b64decode(s)

def b64url_encode(b):
    return base64.urlsafe_b64encode(b).decode().rstrip('=')

# 1) GET /token
# 2) Split and decode
# 3) Tamper payload
# 4) POST to /noverify and /verify
```

Important: JWT signature verification ensures that the token hasn't been tampered with since it was created. The signature is created by combining the header and payload, then encrypting them with a secret key using a specific algorithm. When verifying, the server uses the same secret key to recalculate the signature and compares it to the one in the token. If they match, the token is authentic and trustworthy. If someone modifies the payload without the secret key, the signature won't match and verification fails. This makes JWTs secure for transmitting information between parties because only those with the secret key can create valid tokens. If there's no signature, the JWT uses the "none" algorithm which means it's completely unprotected. Anyone can create or modify these tokens because there's no cryptographic proof of authenticity. This is extremely dangerous in production systems because attackers can forge tokens and impersonate any user. Some JWT libraries have had vulnerabilities where they accept tokens with the "none" algorithm even when they shouldn't. Modern secure applications always reject tokens without signatures or explicitly disable the "none" algorithm. Without a signature, a JWT is just encoded JSON data with no security value at all.

3.6 *JWT authentication bypass via weak signing key.*

(a) Download the newest Docker image:

```
docker pull docker.io/mazurkatarzyna/jwt-ex5
podman pull docker.io/mazurkatarzyna/jwt-ex5
```

(b) Using the prepared Docker image, run simple application that demonstrates weak JWT secret usage:

```
docker run -it -p 5001:5001 --name jwtex5 docker.io/mazurkatarzyna/jwt-ex5
podman run -it -p 5001:5001 --name jwtex5 docker.io/mazurkatarzyna/jwt-ex5
```

If Docker Hub is not responding, use the backup image:

```
docker run -it -p 5001:5001 --name jwtex5 ghcr.io/mazurkatarzynaumcs/jwt-ex5
podman run -it -p 5001:5001 --name jwtex5 ghcr.io/mazurkatarzynaumcs/jwt-ex5
```

Once launched, the application will be available at http://127.0.0.1:5001.

(c) Default credentials are `student / password123` for `student` user. Application has 4 endpoints:

- http://127.0.0.1:5001/vuln/login allows to authenticate as a user and receive a JWT (signed with a weak secret)
- http://127.0.0.1:5001/vuln/admin checks if you are an admin (requires `Authorization: Bearer <token>` and returns the flag if so)
- http://127.0.0.1:5001/fix/login secure version of /vuln/login (uses strong secret)
- http://127.0.0.1:5001/fix/admin secure version of /vuln/admin

(d) Access http://127.0.0.1:5001/vuln/login to generate a JWT for the `student` user (using `student's` username and password). Analyze the received JWT. Try accessing http://127.0.0.1:5001/vuln/admin with the received JWT - did it work?

(e) Using hashcat, John the Ripper or The JSON Web Token Toolkit try to crack JWT's signing secret along with a dictionary of **common JWT secrets**.

(f) After successfully breaking the JWT's singing secret, write a simple script in Python in which you generate the `admin` JWT and gain access to http://127.0.0.1:5001/vuln/admin without knowing the `admin` password.

(g) Try to do the same with /fix/login and /fix/admin endpoints. Did it work?

3.7 *JWT 'none' algorithm and verification vulnerability.*

(a) Using the prepared Docker image, run simple application that demonstrates the JWT 'none' vulnerability:

```
docker run -it --name jwtnone docker.io/mazurkatarzyna/jwt-none
podman run -it --name jwtnone docker.io/mazurkatarzyna/jwt-none
```

If Docker Hub is not responding, use the backup image:

```
docker run -it --name jwtnone ghcr.io/mazurkatarzynaumcs/jwt-none
podman run -it --name jwtnone ghcr.io/mazurkatarzynaumcs/jwt-none
```

(b) Open the `template.py` app inside container, and edit it using `nano` editor:

```
nano template.py
```

Important: With `alg=none`, the token has no signature, allowing anyone to forge tokens with arbitrary payloads that will be accepted if the server permits the `none` algorithm. If there's no signature, the JWT uses the `none` algorithm which means it's completely unprotected. Anyone can create or modify these tokens because there's no cryptographic proof of authenticity. This is extremely dangerous in production systems because attackers can forge tokens and impersonate any user. Some JWT libraries have had vulnerabilities where they accept tokens with the `none` algorithm even when they shouldn't. Modern secure applications always reject tokens without signatures or explicitly disable the `none` algorithm. Without a signature, a JWT is just encoded JSON data with no security value at all.

3.8 **CVE-2024-50634**, **CWE-319** ***Watcharr*** *is an open-source, self-hostable watched list for all your content (movies, tv series, anime) with user authentication, modern and clean UI and a very simple setup. In* ***Watcharr*** *in version* `v1.43.0` *(and below) there exists a vulnerability in a weak JWT token, that allows attackers to perform privilege escalation using a crafted JWT token. This vulnerability is not limited to privilege escalation but also affects all functions that require authentication.*

(a) Using the prepared Docker compose file, run **Watcharr** in the vulnerable version:

```
docker compose -f docker-compose-cve-2024-50634.yml up
podman compose -f docker-compose-cve-2024-50634.yml up
```

Once launched, the application will be available at `http://127.0.0.1:50634`.

If you have problems with running Docker compose file, you can try with the following commands:

```
docker run -p 50634:3080 docker.io/mazurkatarzyna/cve-2024-50634:latest
podman run -p 50634:3080 docker.io/mazurkatarzyna/cve-2024-50634:latest
```

If Docker Hub is not responding, use the backup image:

```
docker run -p 50634:3080 ghcr.io/mazurkatarzynaumcs/cve-2024-50634:latest
podman run -p 50634:3080 ghcr.io/mazurkatarzynaumcs/cve-2024-50634:latest
```

(b) Go to `http://127.0.0.1:50634/` and register first user: `admin`. (First user registered in the application gets the admin permissions.)

(c) Log out of the `admin` account.

(d) Go again to `http://127.0.0.1:50634/` and register a new user with the login `student` and password `student`.

(e) Log out from `student` account.

(f) Log in again as the `admin` user and intercept the request that changes a user's permissions to admin. Do not send the request to the server; simply capture it and forward the captured request to the Burp Repeater.

(g) Log in again as the `student` user and intercept their JWT.

(h) Modify the captured JWT and the previously intercepted request that changes a user's permissions to admin (the request from exercise **3.8**f) in order to assign admin privileges to the `student` account.

(i) Log in again using UI to `http://127.0.0.1:50634/` as the `student` user, and now you should have `admin` permissions.

(j) Click **here** to check out how the authors fixed the vulnerability.

(k) After successfully confirming the existence of the vulnerability, estimate its criticality level using the CVSS **calculator**.

3.9 **CVE-2023-4696**, **CWE-284** *memos is a privacy-first, lightweight note-taking service. Easily capture and share your great thoughts. In version 0.12.2, memos is vulnerable to* ***Improper Access Control****. This vulnerability has the potential to allow any user to modify another user's data, including their password, with relative ease. By exploiting this flaw, attackers could gain unauthorized access to sensitive information, leading to a host of security and privacy concerns. At the heart of this vulnerability is the handling of JSON Web Tokens (JWTs), a commonly used mechanism for secure data exchange. In the memos system, the server fails to adequately verify whether the JWT token has been legitimately issued. This omission means that even a modified JWT, can pass through the system's defenses.*

(a) Using the prepared Docker compose file, run memos in the vulnerable version:

```
docker compose -f docker-compose-cve-2023-4696.yml up
```

Once launched, the application will be available at http://127.0.0.1:4696.

If you have problems with running Docker compose file, you can try with the following commands:

```
docker run -p 4696:5230 docker.io/mazurkatarzyna/cve-2023-4696:latest
podman run -p 4696:5230 docker.io/mazurkatarzyna/cve-2023-4696:latest
```

If Docker Hub is not responding, use the backup image:

```
docker run -p 4696:5230 ghcr.io/mazurkatarzynaumcs/cve-2023-4696:latest
podman run -p 4696:5230 ghcr.io/mazurkatarzynaumcs/cve-2023-4696:latest
```

(b) Knowing that the application is vulnerable to a **Improper Access Control** attack:

- register user in the application, for example `root` (first registered user in the application gets the admin permissions),
- as the `root` user, add 2 users (members) to the application: `student`, `hacker`,
- log in as the `hacker` user,
- as the `hacker` user, knowing that the application uses JWT for authentication, change the password of the `root` (admin) user, thereby preventing the `root` user from accessing the application.
- as the `hacker` user, knowing that the application uses JWT for authentication, try to change the `username`, `email` and the `avatar` of the `student` user

(c) Click **here** to check out how the authors fixed the vulnerability.

(d) After successfully confirming the existence of the vulnerability, estimate its criticality level using the CVSS **calculator**.

3.10 **CWE-798** ***Go LDAP Admin*** *is a OpenLDAP Backend Management Project Implemented in Go and Vue. In version v0.6.0 2025-09-27 it's vulnerable to* ***Use of Hard-coded Credentials****, meaning it has a hardcoded (and weak) JWT signing secret.*

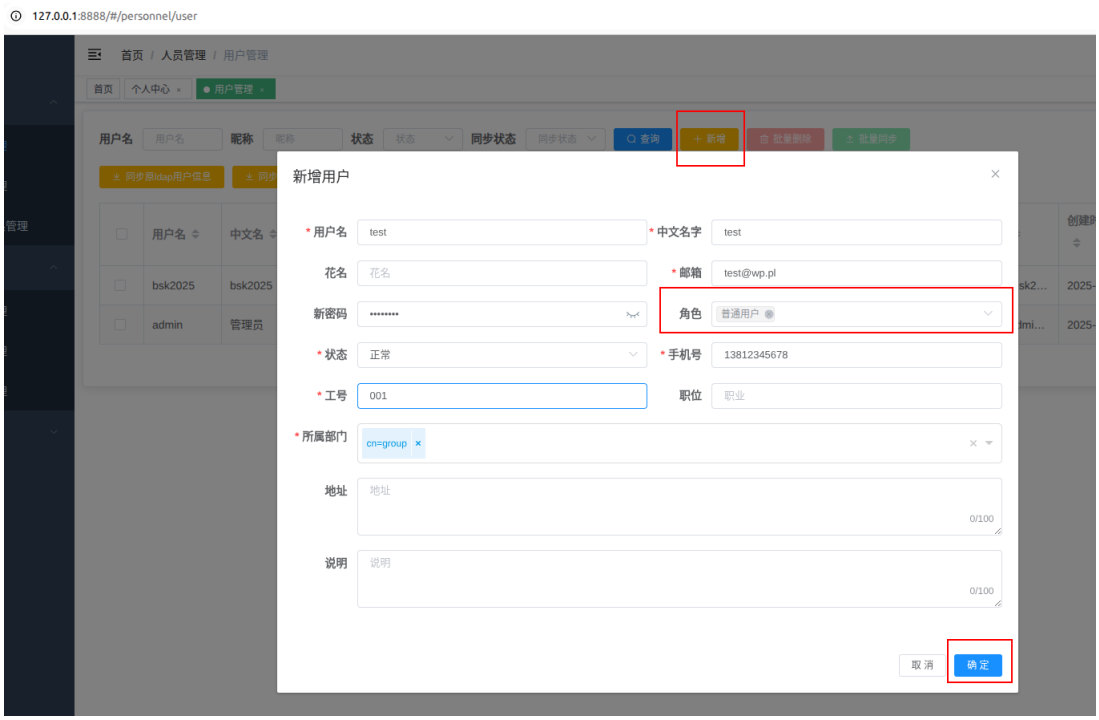
(a) Using the prepared Docker compose file, run **Go LDAP Admin** in the vulnerable version:

```
docker compose -f docker-compose-cwe-798-ex1.yaml up
podman compose -f docker-compose-cwe-798-ex1.yaml up
```

Once launched, the application will be available at http://127.0.0.1:8888.

(b) Log in as `admin` user, credentials are: `admin / 123456`.

(c) Go to http://127.0.0.1:8888/#/personnel/user and add a new, regular user, `student`. Hint:



(d) Log out from `admin` account. Log in as `student` user.

(e) Catch the JWT of the `student` user.

(f) Using **Hashcat**, crack the signing secret in JWT.

(g) Now that we know the default key (signing secret in JWT), let's try to forge the JWT key. The vulnerability is that the JWT only verifies the ID value for users, and the administrator user's ID is 1 by default. Therefore, forge `student's` JWT to create `admin's` JWT. You can use https://www.jwt.io/ or the following code (where `XXX` is the cracked signing secret you cracked with Hashcat):

```
import jwt
import json

def generate_jwt():
    secret = "XXX"
    payload = {
        "exp": 1805610319,
        "identity": 1,
        "orig_iat": 1762410319,
        "user": json.dumps({"ID": 1})
    }

    token = jwt.encode(payload, secret, algorithm="HS256")
    return token

if __name__ == "__main__":
    token = generate_jwt()
    print(f"{token}")
```

To access and run the code, you can use the following Docker container:

```
docker run -it docker.io/mazurkatarzyna/cwe-798-forge:latest
docker run -it ghcr.io/mazurkatarzynaumcs/cwe-798-forge:latest
```

(h) Using forged JWT, try accessing `http://127.0.0.1:8888/api/log/operation/list?username=&ip=&path=&status=&pageNum=1&pageSize=10` using `cURL`.

(i) After successfully confirming the existence of the vulnerability, estimate its criticality level using the CVSS **calculator**.

3.11 **CVE-2025-30204**, **CWE-405** ***golang-jwt*** *is a Go implementation of JSON Web Tokens. Starting in version* `3.2.0` *and prior to versions* `5.2.2` *and* `4.5.2`*, the function* `parse.ParseUnverified` *splits (via a call to* `strings.Split`*) its argument (which is untrusted data) on periods. As a result, in the face of a malicious request whose* `Authorization` *header consists of* `Bearer` *followed by many period characters, a call to that function incurs allocations to the tune of* `O(n)` *bytes (where n stands for the length of the function's argument), with a constant factor of about* `16`*. Successful exploitation of this vulnerability could lead to Denial of Service (DoS) (because too much memory is allocated). This issue is fixed in* `5.2.2` *and* `4.5.2`*.*

(a) Using the prepared Docker image, run a simple server, which uses the **golang-jwt** library in the vulnerable version:

```
docker run -dp 30204:30204 --name cve202530204 docker.io/mazurkatarzyna/cve-2025-30204:latest
podman run -dp 30204:30204 --name cve202530204 docker.io/mazurkatarzyna/cve-2025-30204:latest
```

Once launched, the application will be available at `http://127.0.0.1:30204`.

(b) Since the vulnerability is related to allocating a large amount of memory, first open a new console window and enable monitoring of the container's resource usage, including memory consumption:

```
docker stats cve202530204
podman stats cve202530204
```

(c) Server is using the vulnerable `parse.ParseUnverified` function to parse JWT given in the `Authorization` headerm, and return it's parsed content to the user. Send a simple `GET` request to server using `cURL`. Add only `Authorization` header. (You can generate a sample JWT for the `Authorization` header at `https://www.jwt.io/`).

(d) Send a `GET` request to the server using `cURL` with a crafted JWT to cause a DoS attack.

(e) Click **here** to check out how the authors fixed the vulnerability.

(f) After successfully confirming the existence of the vulnerability, estimate its criticality level using the CVSS **calculator**.

Links

- https://www.jwt.io/
- https://datatracker.ietf.org/doc/html/rfc7519
- https://www.jwt.io/introduction#what-is-json-web-token
- https://pyjwt.readthedocs.io/en/stable/
- https://owasp.org/www-project-web-security-testing-guide/latest/JWT
- https://cheatsheetseries.owasp.org/cheatsheets/JWT
- https://portswigger.net/web-security/jwt
- https://github.com/wallarm/jwt-secrets
- https://github.com/ticarpi/jwt_tool
- https://pentesterlab.com/blog/another-jwt-algorithm-confusion-cve-2024-54150
- https://auth0.com/blog/brute-forcing-jwt/