> Disclaimer
> This write-up is for educational purposes only.
> The challenge was solved in a legal CTF environment.
> No real-world systems were targeted.

Semicolon CTF Challenge 2 – Full Write-Up

> Disclaimer
> This write-up is for educational purposes only.
> The challenge was solved in a legal CTF environment.
> No real-world systems were targeted.

Challenge Overview

Name: Semicolon – CTF Challenge 2
Category: Web / Container / Reverse Engineering
Difficulty: Medium
Goal: Find the hidden flag

The challenge presents a modern task management web application built with **Next.js** and deployed using **Docker**. At first glance, the application appears harmless, but deeper inspection reveals obfuscated runtime logic that leads to the flag.

Environment Setup

- Linux environment
- Docker
- Node.js 20 (containerized)
- Python 3 (for decoding)

The challenge was extracted and run locally.

🔍 Step 1: Exploring the Application

After running the application, the web interface loaded successfully on:

http://localhost:3000

The dashboard showed pre-defined tasks, but none of the UI features exposed the flag directly. This suggested the flag was hidden **server-side**.

Step 2: Inspecting the Docker Setup

The Dockerfile revealed that the container starts via a custom script:

CMD ["/app/scripts/init.sh"]

This made init.sh a high-value target for analysis.

Step 3: Analysing init.sh

The scripts/init.sh file contained:

- Heavy shell obfuscation
- Multiple anonymous functions
- Hex encoding
- XOR operations

Key observations:

- The script reads two hidden files:
  - scripts/data/.session
  - scripts/data/.runtime
- Their contents are processed and written to a decoded output file.

This strongly suggested **manual decoding was required**.

Step 4: Investigating Hidden Runtime Files

Listing the directory:

ls -la scripts/data

Revealed:

- .session
- .runtime

- .env.production
- .cache

Reading the relevant files:

cat scripts/data/.session
cat scripts/data/.runtime

- .session → encrypted hex string
- .runtime → long hex key

Step 5: Reversing the Obfuscation

By understanding the shell functions, I determined that:

- Each byte of .session is XORed with the corresponding byte in .runtime
- The result is converted back to ASCII characters

To replicate this safely, I wrote a short Python script:

```
s =
"efb486919ec690c682a4fbcbcfc589b89cc7cf83ddbef4e9c1c790e1b3819d94f6acd1c092c28f8fbb
db948b"
k =
"a9f8c7d6e5b4a3f2e1d0c9b8a7f6e5d4c3b2a1f0e9d8c7b6a5f4e3d2c1b0a9f8c7d6e5b4a3f2e1d0c
9b8a7f6e5d4c3b2a1f0"

out = ""
for i in range(0, len(s), 2):
    out += chr(int(s[i:i+2], 16) ^ int(k[i:i+2], 16))

print(out)
```

Step 6: Flag Recovered

Running the script revealed the flag:

FLAG{r34ct2sh3ll_uns4f3_d3s3r14l1z4t10n_rc3}

▶ Final Flag

FLAG{r34ct2sh3ll_uns4f3_d3s3r14l1z4t10n_rc3}

Key Takeaways

- Obfuscated shell scripts can hide critical logic
- Docker init scripts are common CTF attack surfaces
- XOR + hex encoding is a classic but effective hiding technique
- Understanding execution flow is more important than the UI

Conclusion

This challenge combined **container analysis**, **shell reverse engineering**, and **basic cryptographic decoding**. By focusing on the runtime logic instead of the frontend, the flag was successfully extracted.

Author: Safa Imad.
Status: Solved ✓
Platform: Semicolon CTF