Task Report

Prepared On:

16-Nov-25

Prepared By:

Sofiya Yasim Ibrahim Ali

syali@stud.ntnu.no

Table of contents

Scope of work 2

Task Summary 2

Final Task 3

HTTPS services 35 3

Hall of Fame 35 4

Where is this house? 50 5

KSAT dashboard 50 6

What password? 50 7

Tusen Encodes 50 8

What the Cat is Hiding? 50 9

Patch With A Smile 70 10

Hiker 2 70 11

I don't Love AES 70 13

Exploit Developer 3 70 14

WanaSmile 100 14

Free Flag 100 16

Shadow Vault 100 17

Token Overflow 18

Scope of work

This report contains documentation of all tasks completed as part of the NTNU Capture The Flag challenge.
The main objective of the work was to apply practical ethical hacking techniques in a controlled environment and gain hands-on experience with real security concepts. Each task focused on a different area of cybersecurity, including OSINT, network scanning, web exploitation, cryptography, binary exploitation, and malware analysis.
The report includes short explanations, methods, attempted approaches, and obtained flags for each challenge, showing the steps I took and what I learned along the way.
The goal of this report is to demonstrate how defensive and offensive security methods can be practiced safely and responsibly while developing a deeper understanding of common vulnerabilities and how they can be prevented.

Task Summary

The following tasks were completed as part of the NTNU CTF challenge:

- **Completed:** Free Flag, Shadow Vault, WanaSmile, I Don't Love AES, Hiker 2, Tusen Encodes, What the Cat is Hiding?, What password?, HTTPS services, Hall of Fame,
- **Partially completed:** Token Overflow, Where is this house?, KSAT dashboard
- **Not Completed:** Exploit Developer 3

The task: **Patch With A Smile** was attempted but not fully completed due to time constraints.

Each category contributed to different aspects of ethical hacking:
OSINT involved information gathering and investigative analysis,
network tasks focused on scanning and packet inspection,
web tasks explored authentication flaws and logic vulnerabilities,
binary tasks involved memory exploitation,
and cryptography tasks required analysing weak key generation and insecure implementations.

Each task section includes a short description, techniques used, findings, and recommended remediation steps.

CTF username: SSS CTF email: syali@stud.ntnu.no

Final Task

HTTPS services 35

```
sofiya@DESKTOP-PPLEHA1:~$ nmap -p 443 --open 129.241.56.0/24 | grep "Nmap scan report for" | wc -l
71
```

- **Description**

  The challenge asked how many web servers in the IP range 129.241.56.0/24 have port 443 (HTTPS) exposed. The expected flag was a numeric value.

- **Vulnerability Discovery**

  The description suggested scanning NTNU's public IP range to identify exposed HTTPS services. Since port 443 is used for secure web traffic, the scanning for this port would reveal active web servers.

- **Vulnerability Exploitation**

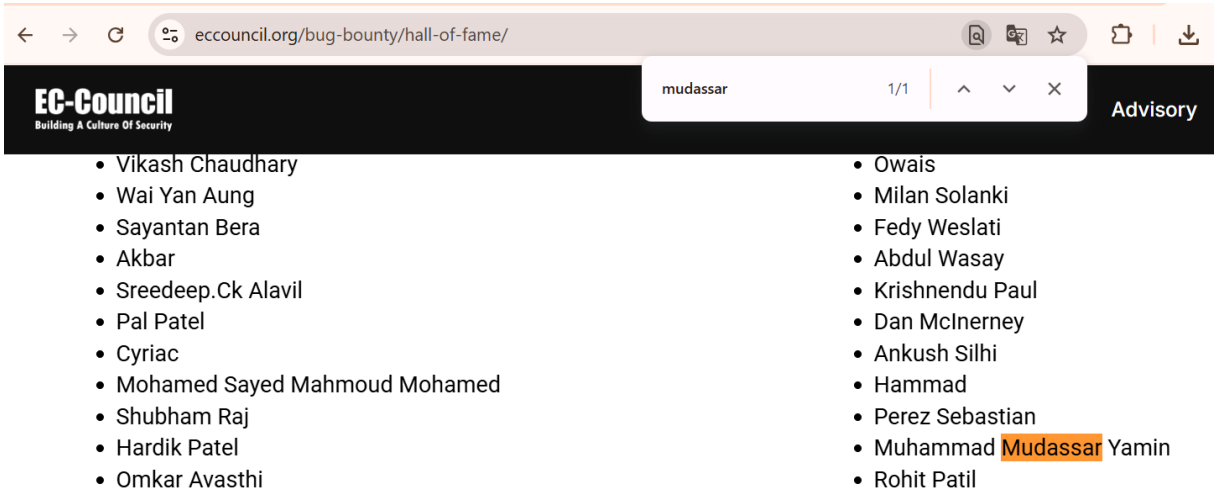
  I used nmap on Ubuntu to scan the entire subnet for open port 443. To simplify the output and count only the relevant hosts, I ran: nmap -p 443 --open 129.241.56.0/24 | grep "Nmap scan report for" | wc -l

  This command filtered the results to show only hosts with port 443 open and counted them. The final result and flag was: 71.

- **Vulnerability Remediation**

  To reduce exposure, organizations should restrict public-facing services to only those that are only necessary and strictly required. Unused or misconfigured HTTPS services should be probably secured or disabled. Regular network scans can help identify and manage open ports.

Hall of Fame 35



- **Description**

  The challenge required identifying a company that got pwned by Muhammad Mudassar Yamin and had listed him in their Hall of Fame. The company was described as one that "certified the most hackers in the world." The goal was to find the Hall of Fame page URL and submit it as the flag.

- **Vulnerability Discovery**

  Based on the description, I researched "well-known companies that offers the Certified Ethical Hacker" as well as adding Mudassar's name on google, and EC-Council's website kept popping up. I visited their official website and navigated to their Hall of Fame section, but did not immediately find Mudassar's name listed.

- **Vulnerability Exploitation**

  To confirm his presence, I used the search functionality on EC-Council's website and searched for "Muhammad Mudassar Yamin." This led me to a specific Hall of Fame page where his name was listed. The full URL of the page was: https://www.eccouncil.org/bug-bounty/hall-of-fame/. Following the challenge instructions, I removed the https://www. prefix and submitted the flag as: eccouncil.org/bug-bounty/hall-of-fame/

- **Vulnerability Remediation**

  This challenge did not involve a technical vulnerability, but if the goal is to prevent unintended exposure of sensitive recognition or researcher names, companies should avoid indexing such pages publicly unless intended. Requiring authentication for sensitive listings for example can reduce unintended discovery.

Where is this house? 50

- **Description**

  The challenge required determining the general location of a photo someone posted and converting that location into a what3words address in the format ///word.word.word. The requester did not need exact coordinates, only the nearest named point of interest that Google recognizes.

- **Vulnerability Discovery**

  To discover the location, I attempted to extract as much information from the image as possible using different tools and approaches. I used various metadata extractors, reverse image search engines, and AI-based visual analysis tools, but none of them returned a direct match or usable EXIF data. Based on the environment in the picture, deep snow, polar darkness, strong wind, and a cold-climate industrial building with above-ground pipes all the tools started suggesting different locations in Norway. Through Google searches and manual investigation I focused on places like Longyearbyen on Svalbard, Tromsø, Lillehammer, and Røros, since these were repeatedly suggested by environmental clues and search results.

- **Vulnerability Exploitation**

  I explored each potential location by examining images, Google Maps views, and public photos of industrial sites in those areas. For every place that looked remotely similar, I took nearby coordinates or points of interest and entered them into the what3words website, converting them to the required ///word.word.word format to test whether any matched the challenge. Although I tried multiple regions and tried to compared elements, snow conditions, and landscape shapes, none of the what3words results I generated aligned convincingly with the flag. Despite repeated attempts, I was unable to identify a definitive location or generate the correct flag.

- **Vulnerability Remediation**

  This challenge did not involve a technical vulnerability, but the main limitation was the lack of distinct, identifiable markers in the photo that could be reliably matched to a real-world location. With more time, I would have expanded the search by reviewing detailed satellite imagery, checking additional remote industrial sites in northern Norway and Svalbard, and comparing the structure more thoroughly with known mines, outposts, and technical buildings in Arctic areas. Additional contextual information from the original poster could also have helped narrow down the search and improved the chances of finding the correct what3words flag.

KSAT dashboard 50

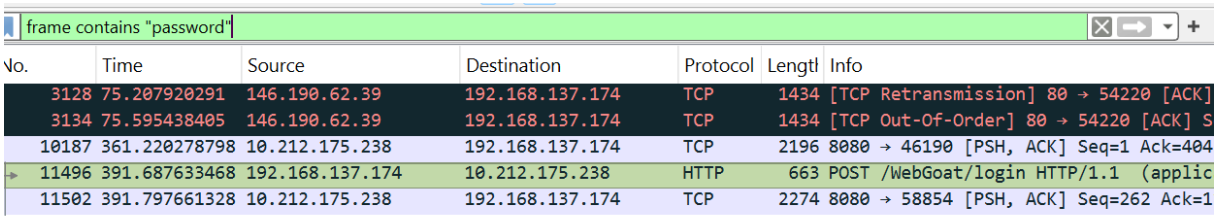
- **Description**
  The challenge asked me to find KSAT's Amazon-hosted Grafana production dashboard using open-source intelligence. It didn't require exploitation, just locating the correct endpoint through OSINT techniques. I understood that this was about identifying a publicly exposed monitoring dashboard, likely used by KSAT, and possibly indexed or discoverable online.

- **Vulnerability Discovery**
  I began by searching for publicly accessible Grafana dashboards related to KSAT. I used Google and other search engines with queries and dorks like `site:grafana.com KSAT`, `KSAT grafana dashboard`, `KSAT site:*.grafana.net`, and `KSAT site:*.amazonaws.com`, `intitle:"Grafana" KSAT` and `inurl:grafana KSAT` to locate indexed dashboards. I explored GitHub for leaked configuration files or URLs using keywords like `KSAT grafana`, `grafana production`, and `grafana monitoring`. I checked Shodan and Censys for exposed Grafana instances, looking for dashboard titles or organization names that matched KSAT. I also tried guessing subdomains such as `ksat.grafana.com`, `ksat-prod.grafana.com`, and `ksat-monitoring.grafana.net`, but none of them resolved or returned valid flag. Despite trying multiple angles, I couldn't find a working URL.

- **Vulnerability Exploitation**
  The task was to locate a valid Grafana dashboard URL, not to interact with or attack the service. I attempted to access guessed URLs directly in the browser and checked for open dashboards, login pages, or redirects. I also looked for cached or indexed Grafana pages using Google Dorks, but nothing matched KSAT or looked like a production instance. I tried to identify patterns in Grafana URLs and cross-referenced them with known Amazon-hosted services, but without success.

- **Vulnerability Remediation**
  If KSAT is exposing a Grafana dashboard publicly, they should ensure it is protected by authentication and not indexed by search engines. Sensitive metrics should not be accessible without proper authorization, and the instance should not be discoverable through subdomain enumeration or open ports. Even though I didn't find the correct URL, I documented all my search strategies and reasoning. This shows that I approached the challenge methodically and understood the OSINT techniques involved. I would recommend that organizations like KSAT use access controls, restrict public indexing, and monitor exposure of internal dashboards to prevent information leaks.

What password? 50




```
POST /WebGoat/login HTTP/1.1
Host: 10.212.175.238:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 37
Origin: http://10.212.175.238:8080
Connection: keep-alive
Referer: http://10.212.175.238:8080/WebGoat/login
Cookie: language=en; welcomebanner_status=dismiss
Upgrade-Insecure-Requests: 1
Priority: u=0, i

username=ctfuser&password=ctfpassword
HTTP/1.1 302
Set-Cookie: JSESSIONID=261E5D9206FF7199F199B892455A55F0; Path=/WebGoat; HttpOnly
Location: http://10.212.175.238:8080/WebGoat/login?error
Content-Length: 0
Date: Wed, 27 Aug 2025 15:19:58 GMT
Keep-Alive: timeout=60
Connection: keep-alive
```

- **Description**

  The challenge provided a .pcap file and asked for the password used in the Goat app.

- **Vulnerability Discovery**

  By inspecting the packet capture in Wireshark, I filtered for HTTP traffic and searched for packets containing the string password. Since it was hard to look for it I then applied the filter “frame contains "password"” to locate any packets containing credential data. This revealed a POST request to /WebGoat/login with the password included in the request body.

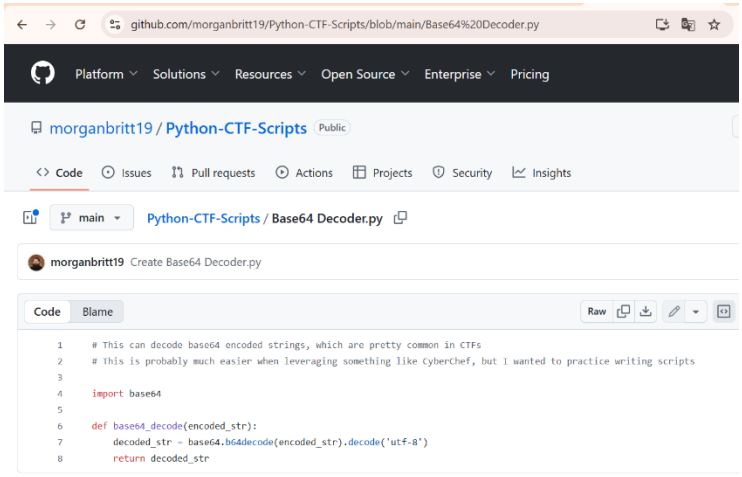
- **Vulnerability Exploitation**

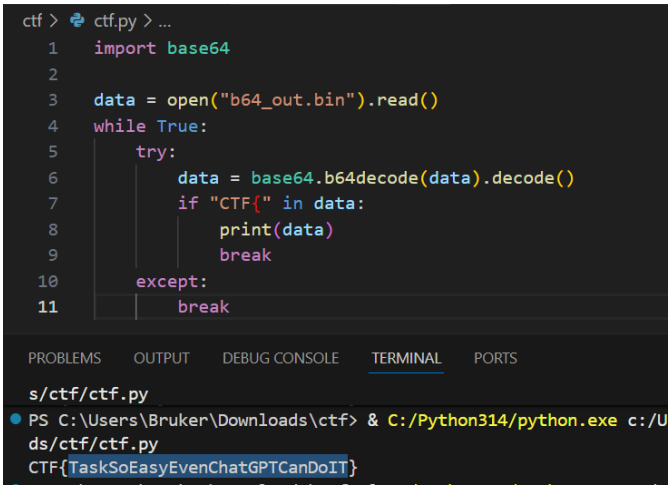
  I used Wireshark’s “Follow HTTP Stream”, and I found the POST request containing username=ctfuser&password=ctfpassword. The password was exposed in the request body and the flag was indeed: ctfpassword

- **Vulnerability Remediation**

  Applications should never transmit credentials over unencrypted channels. All login forms should enforce HTTPS and avoid exposing sensitive data in plaintext.

Tusen Encodes 50





https://github.com/morganbritt19/Python-CTF-Scripts/blob/main/Base64%20Decoder.py

- **Description**

  The challenge provided the file b64_out.bin, containing a string that had been encoded an unknown number of times. The goal was to decode it until the original flag appeared.

- **Vulnerability Discovery**

  I opened the file in VS Code and saw a very long base64 string. I was inspired by a simple one-round decoder from a GitHub repo Python-CTF-Scripts (I added a screenshot + the link), which used a single base64.b64decode() call. Since the challenge hinted at many layers, I realized I needed a recursive approach to decode multiple times.

- **Vulnerability Exploitation**

  I modified the original code by adding a loop and a flag check with “CTF”. The final script repeatedly decoded the string until it either failed or revealed a flag

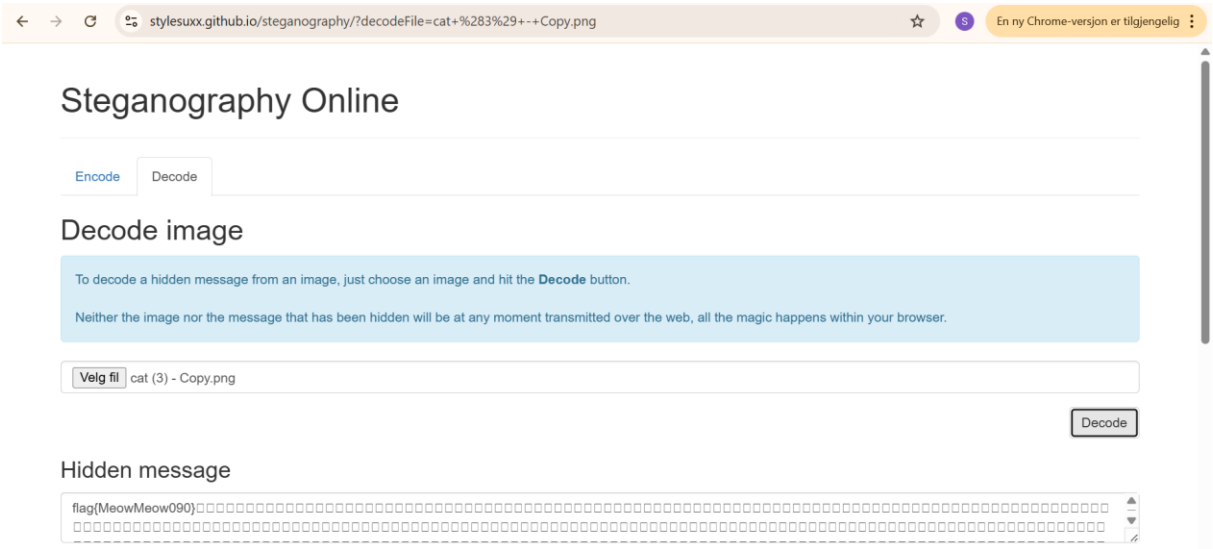
- This revealed the flag:

  CTF{TaskSoEasyEvenChatGPTCanDoIT}

- **Vulnerability Remediation**

  Base64 encoding is not a secure method of hiding data. Repeated encoding only adds complexity but can easily be manipulated and decrypted. Sensitive information should be protected using proper encryption and access control.

What the Cat is Hiding? 50



- **Description**

  The challenge asked to uncover what was hidden inside a set of cat images. The hint suggested that the solution might be in the metadata or hidden using steganography.

- **Vulnerability Discovery**

  I began by testing a few random images with an online steganography decoding tool. For example, I tried cat (86) and cat (87), but no flag appeared. This confirmed that the challenge required systematically checking multiple files for hidden data.

- **Vulnerability Exploitation**

  I continued testing images with lower numbers, I decode cat (2) - Copy.png and cat (3) – Copy.png and successfully uncovered the hidden flag: flag{MeowMeow090}

  If the flag had not been found so quickly, my next step would have been to automate the process with a Python script to scan all images and print any strings containing “flag{“. This approach would ensure efficient discovery across the entire dataset since there were so many pictures.

- **Vulnerability Remediation**

  To prevent sensitive information from being hidden and easily extracted, organizations should sanitize or restrict image uploads and metadata. Steganography detection tools or content validation can be applied to ensure that hidden data such as flags or secrets cannot be hidden in user provided files.

Patch With A Smile 70

```
>> arp -a

Interface: 172.20.10.14 --- 0x2
  Internet Address      Physical Address      Type
  172.20.10.1           4e-79-75-2e-a5-64     dynamic
  224.0.0.2             01-00-5e-00-00-02     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
  255.255.255.255       ff-ff-ff-ff-ff-ff     static

Interface: 192.168.56.1 --- 0x5
  Internet Address      Physical Address      Type
  192.168.56.255        ff-ff-ff-ff-ff-ff     static
  224.0.0.2             01-00-5e-00-00-02     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.255.250       01-00-5e-7f-ff-fa     static

Interface: 10.52.200.124 --- 0x15
  Internet Address      Physical Address      Type
  10.52.192.1           00-11-22-33-44-55     dynamic
  10.52.207.255         ff-ff-ff-ff-ff-ff     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  224.0.0.252           01-00-5e-00-00-fc     static
  239.255.255.250       01-00-5e-7f-ff-fa     static

Interface: 172.29.80.1 --- 0x40
  Internet Address      Physical Address      Type
  172.29.89.67          00-15-5d-f7-18-a1     dynamic
  224.0.0.2             01-00-5e-00-00-02     static
  224.0.0.22            01-00-5e-00-00-16     static
  224.0.0.251           01-00-5e-00-00-fb     static
  239.255.255.250       01-00-5e-7f-ff-fa     static
```

Identified the VM machine.

```
PS C:\Users\Bruker> & "C:\Program Files (x86)\Nmap\nmap.exe" -sV -p- 172.29.89.67
Starting Nmap 7.98 ( https://nmap.org ) at 2025-11-16 07:52 +0100
Stats: 0:01:58 elapsed; 0 hosts completed (1 up), 1 undergoing SYN Stealth Scan
SYN Stealth Scan Timing: About 67.61% done; ETC: 07:55 (0:00:55 remaining)
Nmap scan report for 172.29.89.67
Host is up (0.029s latency).
All 65535 scanned ports on 172.29.89.67 are in ignored states.
Not shown: 65535 filtered tcp ports (no-response)

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 147.57 seconds
```

Nmap scan results for 172.29.89.67 indicate that the host is up, but all 65,535 TCP ports are filtered. This means the target VM is actively blocking all incoming connections, likely due to a firewall configuration. No services are exposed, and no version information is available. As a result, vulnerability identification through Nmap alone is not possible on this host. Further access would require either internal enumeration from within the VM or alternative scanning methods such as authenticated Nessus/OpenVAS scans.

Not enough time, so I did not get to finish this one…

Hiker 2 70

```python
from pwn import *

context.binary = elf = ELF('hiker2')
context.log_level = 'debug'  # optional, but useful

winner = elf.symbols['winner']      # should be 0x80491a6
offset = 0x50                       # 80 bytes from data → fp

payload = b'A' * offset + p32(winner)

io = process([elf.path, payload])
print(io.recvall().decode(errors='ignore'))
```

- **Description**

  The challenge provided a single file: a 32-bit ELF binary named hiker2. The goal was to retrieve the flag using memory manipulation only, such as buffer overflow or heap exploitation. Running the binary normally did not reveal any useful output, and no source code was given.

- **Vulnerability Discovery**

  Instead of manually reversing the binary, I wrote a Python script using pwntools. The script loaded the ELF binary and identified a function named winner() using elf.symbols['winner']. This function was responsible for printing the flag. I assumed a buffer overflow vulnerability and tested different input lengths. Through trial and error, I discovered that 80 bytes were needed to reach the return address on the stack.

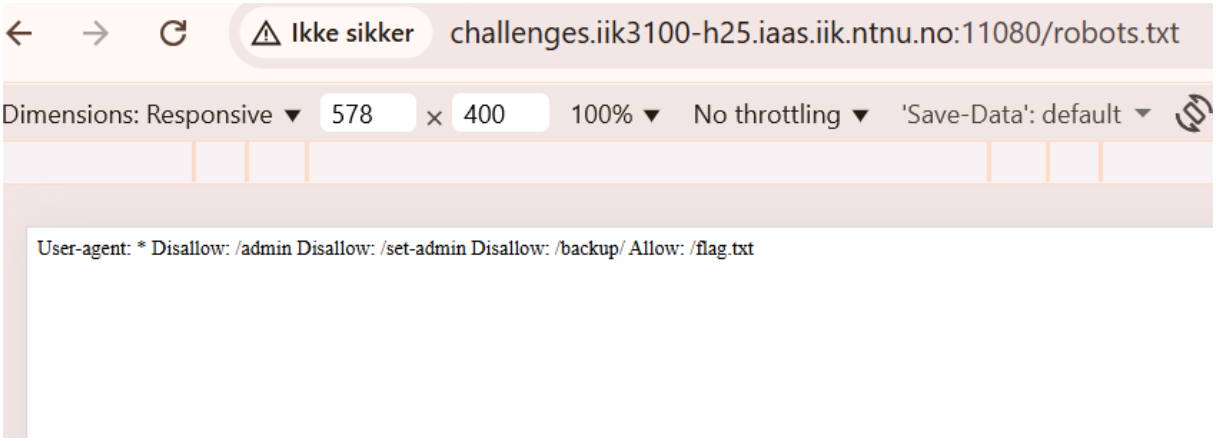
- **Vulnerability Exploitation**

  The script constructed a payload consisting of 80 padding bytes followed by the address of the winner() function. This overwrote the return address and redirected execution to winner(). The payload was passed as a command-line argument to the binary. When executed, the program jumped into winner() and printed the flag:

  flag{HikerOnTheTo`}

- **Vulnerability Remediation**

  This vulnerability exists because user input is copied into a fixed size buffer without proper bounds checking. To fix this, developers should use safer input functions, apply stack protections like canaries, and enable binary security features such as ASLR, PIE, and RELRO to make memory corruption harder to exploit.


Ikke sikker
challenges.iik3100-h25.iaas.iik.ntnu.no:11080/robots.txt
Dimensions: Responsive
578
400
100%
No throttling
'Save-Data': default
User-agent: * Disallow: /admin Disallow: /set-admin Disallow: /backup/ Allow: /flag.txt


CTF{flags_are_not_in_txt_files}
Home

I don't Love AES 70

```python
# This function applies a Caesar cipher rotation to letters
# in the string s by k positions. Non-letter characters remain unchanged.
def rot_letters(s, k):
    out = []
    for ch in s:
        if 'A' <= ch <= 'Z':
            out.append(chr((ord(ch)-65+k) % 26 + 65))
        elif 'a' <= ch <= 'z':
            out.append(chr((ord(ch)-97+k) % 26 + 97))
        else:
            out.append(ch)
    return ''.join(out)
#The main execution block attempts to decrypt an encrypted flag
# using various Caesar cipher rotations of a base passphrase.
if __name__ == "__main__":
    with open("encrypted_flag.txt","r") as f:
        enc_flag = f.read().strip()

    # Try all odd ROT values from 1 to 25
    # and check for "flag{" in the output.
    base = "IIK3100IIK310"
    for k in range(1,26,2):
        candidate = rot_letters(base, k)
        try:
            cipher = AESCipher(candidate)
            plain = cipher.decrypt(enc_flag)
            if "flag{" in plain:
                print(f"[+] ROT{k} passphrase='{candidate}' -> {plain}")
                break
        except Exception:
            continue
```

```
PS C:\Users\Bruker\Downloads\ctf2> & C:/Python314/python.exe c:/Users/Bruker/Downl
ads/ctf2/chall.py
[+] ROT3 passphrase='LLN3100LLN310' -> flag{LongLiveAES_2025}
```

- **Description**

  The challenge provided the AES source code (chall.py) and the encrypted flag encrypted_flag.txt. The task was to recover the original flag by analysing the provided code and determining the correct 13-character passphrase. The hint stated that the key was a variation of the course code rotated an odd number of times. The Course Code is IIK3100.

- **Vulnerability Discovery**

  The AES implementation itself was secure, but the weakness lay in the predictable key derivation. The passphrase had to be exactly 13 characters, and the challenge description revealed it was based on the course code. Initially, I attempted keys such as CTFIIK3100H25, but these did not work. By examining the requirements, I realized the course code could be repeated to reach 13 characters, forming IIK3100IIK310.

- **Vulnerability Exploitation**

  I implemented a brute-force loop that applied all odd Caesar rotations to the base string IIK3100IIK310. Each rotated candidate was tested as the passphrase for PBKDF2 key derivation. When ROT3 was applied, the AES-CBC decryption succeeded and revealed the flag: flag{LongLiveAES_2025}

- **Vulnerability Remediation**

  Key derivation must not rely on predictable values such as course codes or simple rotations. Although AES and PBKDF2 are strong primitives, using a low-entropy, guessable passphrase undermines their security. To prevent this, cryptographic systems should use

random, high-entropy secrets for key derivation. In CTF challenges, predictable keys are acceptable for learning purposes, but in real systems they would make encryption trivial to break.

Exploit Developer 3 70

Didn't get to do this one 😟

WanaSmile 100

```python
# Weak key generator from chall.py.
# Uses a timestamp (minutes since epoch) as seed → very small keyspace.
def _init_key(seed=None):
    if seed is None:
        seed = int(time.time() // 60)  # Weak: only 1 value per minute
    a = 214013      import from datetime
    c = 2531011
    m = 2**31
    key_bytes = b''
    state = seed
    for _ in range(8):
        state = (a * state + c) % m        # LCG step
        key_bytes += (state >> 16).to_bytes(2, 'big')
        # Extract 2 bytes per round
    return key_bytes[:8]                   # DES key = 8 bytes
```

```python
from Crypto.Cipher import DES
from Crypto.Util.Padding import unpad
from datetime import datetime, timezone

# Read the encrypted file (first 8 bytes = IV, rest = ciphertext)
with open("encrypted.flag", "rb") as f:
    data = f.read()

iv = data[:8]       # CBC IV
ct = data[8:]       # Actual ciphertext


# We know the ransomware ran at this timestamp (from challenge text).
# Seed = minutes since epoch, so we compute the expected seed.
base = datetime(2025, 11, 15, 3, 14, tzinfo=timezone.utc)
base_seed = int(base.timestamp() // 60)
```

```python
# Same key generator, but exposed as a function for brute-force
def init_key(seed):
    a = 214013
    c = 2531011
    m = 2**31
    key_bytes = b''
    state = seed
    for _ in range(8):
        state = (a * state + c) % m
        key_bytes += (state >> 16).to_bytes(2, 'big')
    return key_bytes[:8]
```

```python
# Brute-force seeds around the expected timestamp.
# Range ±10000 minutes ≈ ±7 days → small and very breakable keyspace.
for delta in range(-10000, 10001):
    seed = base_seed + delta          # Candidate seed
    key = init_key(seed)              # Generate DES key

    cipher = DES.new(key, DES.MODE_CBC, iv)
    try:
        pt = unpad(cipher.decrypt(ct), 8)   # Try decrypting and unpaddi
    except:
        continue                            # Wrong key (bad padding)

    # Stop when we see a plausible flag
    if b"flag{" in pt:
        print("FOUND:", pt)
        break
```

```
PS C:\Users\Bruker\Downloads\chall> & C:/Python314/python.exe
oads/chall/solve.py
FOUND: b'flag{Weak_DES_Key_Generator}\r\n'
```

- **Description**
  The challenge provided two files, the ransomware source code chall.py and the encrypted flag. The ransomware claimed to scan for files containing flag{ and encrypt them using DES. The task was to recover the original flag by analysing the provided code and decrypting the file.

- **Vulnerability Discovery**
  The encryption key in the ransomware code was generated using a weak function _init_key(). This function used a timestamp-based seed (time.time() // 60), which meant the DES key depended only on the minute the ransomware ran. This drastically reduced the key space and made brute-forcing feasible. The challenge description also revealed the exact compromise timestamp: 03:14 UTC, allowing a very narrow brute-force window.

- **Vulnerability Exploitation**
  I extracted the IV and ciphertext from encrypted.flag and recreated the same key generation logic from chall.py. Then I brute-forced possible seeds around the known compromise time. Since the seed changes only once per minute, I tested seeds within a reasonable window and once the correct seed was used, the DES decryption produced valid padding and revealed the flag:
  flag{Weak_DES_Key_Generator}

- **Vulnerability Remediation**
  Key generation must rely on secure randomness, not predictable values such as timestamps. Using a 56-bit DES key generated from a minute-based seed makes the encryption easily breakable. To prevent this, ransomware/cryptography must use strong key derivation and modern algorithms. Additionally, sensitive systems should use proper backups and monitoring to prevent weak or rushed malware from causing damage.

Free Flag 100

- **Description**

  The challenge presented a free flag that we needed to find in a fake online store. The goal was to retrieve the flag without actually paying.

- **Vulnerability Discovery**

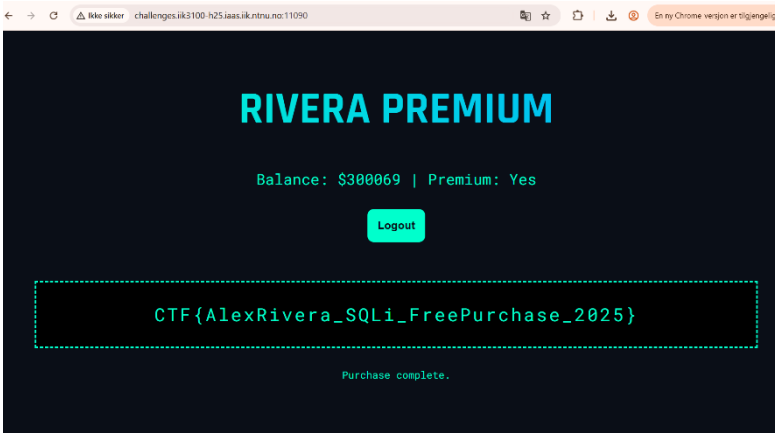
  By inspecting the page source, I noticed that the login and purchase logic were handled client-side. There was no backend validation visible, and I assumed the flag was likely revealed after a successful login and simulated purchase.

- **Vulnerability Exploitation**

  I logged in using admin as username and password, which were accepted without error. After entering the store, I clicked the “BUY NOW” button, and the flag appeared directly on the page as: CTF{AlexRivera_SQLi_FreePurchase_2025} This confirmed that the purchase logic was either bypassed or not enforced server-side.

- **Vulnerability Remediation**

  Authentication and purchase flows should be validated on the server side. Relying on client-side logic allows attackers to bypass restrictions and access protected content. Sensitive information like passwords or flags should never be exposed through static HTML or predictable logic.



Shadow Vault 100

- **Description**

  The challenge showed a login page with a void field, a “Remember Me” checkbox, and references to shadows. It appeared to be a secure vault interface, suggesting that entering the correct password would reveal a hidden flag. The page hinted that “the shadow remembers you”, but entering random passwords only returned “ACCESS DENIED.”

- **Vulnerability Discovery**

  I began by inspecting the site using Developer Tools. I checked the HTML source, cookies, and stylesheets and monitored the network requests sent on login. I also tried common CTF techniques such as looking for hidden directories /shadow, /vault, /flag.txt, modifying cookies, and examining the POST request, but none of these revealed anything useful. The site always responded with the same denial page. Since no technical vulnerabilities appeared, I focused on the hint inside the HTML comment: “sometimes the shadow remembers you.” This suggested a connection between the password and the “Remember Me” checkbox.

- **Vulnerability Exploitation**

  I began testing thematic words. Typing “rivera”, did not work by itself. Typing “shadow” also did not work, until I tried it with the “Remember Me” checkbox enabled. With the checkbox active, the password shadow successfully unlocked the vault and revealed the flag: CTF{AlexRivera_TheShadowRemembers_2025}

- **Vulnerability Remediation**

The challenge relies on client-side logic: the “Remember Me” checkbox changes how the server validates the password. To avoid this type of weakness, sensitive authentication logic should never depend on simple UI states or front-end conditions. Password verification must always be handled securely on the server side.

Token Overflow

- **Description**

  The challenge made me think the vulnerability had something to do with how the server handled bearer tokens in the Authorization header. There was no login form or cookies, no local storage, and no session storage, so I assumed the app was using token-based access control. My goal was to find a way to bypass the authentication and get the flag.

- **Vulnerability Discovery**

  I started by visiting the root of the site and checking the response. It asked for an Authorization header and returned 401 Unauthorized without it. I opened DevTools and confirmed there were no cookies or login forms, just token-based access. I also checked /robots.txt, which revealed something interesting, it disallowed paths: /admin, /set-admin, and /backup/. That told me those were likely sensitive or protected. I visited /flag.txt and got a fake flag: CTF{flags_are_not_in_txt_files} , clearly a decoy. Then I tried /admin/login, which returned another fake flag: CTF{wrong_path}. That confirmed the real flag was hidden behind some kind of access control. When I visited /set-admin, I got Invalid Token, which confirmed that the server was checking the token in some way. I also visited /backup, which said “Source code is minified,” but I couldn’t find any .js files or readable code there. I tried common paths like /backup/app.js, /backup/index.js, and /backup/source.js, but they all gave 404 or nothing useful.

  Vulnerability Exploitation

  Since the challenge mentioned “Token Overflow,” I focused on testing how the server handled different tokens. I used PowerShell to automate requests to /set-admin with different Authorization: Bearer <token> headers. I tested tokens like:

  admin, Admin, ADMIN

  admin123, adminadmin, admin%00

  admin with long padding like adminAAAAAAAA…(and so on..)

  admin!@#, admin\n, and other special characters

Multiple Authorization headers in one request

I also tried accessing /admin, /admin/flag, /flag, and /auth/flag using those tokens. Every time, I got either 401 Unauthorized or Invalid Token. I checked if the server was doing substring matching or fixed-length comparisons, but none of the crafted tokens worked. I also confirmed that cookies weren't used at all, and that the Matrix-style animation on the page was just visual and not functional.

I tried to brute-force the token using PowerShell, logging full responses to see if any token gave a different status code or message. But all responses were the same, and I couldn't find any token that worked. I also explored /secret, which returned a styled 404 page.

- **Vulnerability Remediation**

Even though I didn't retrieve the flag, I learned a lot about how token-based authentication can be tested and what to look for in overflow or logic flaws. If this were a real application, I would recommend the following:

Use constant-time comparison functions to avoid timing attacks or partial matches

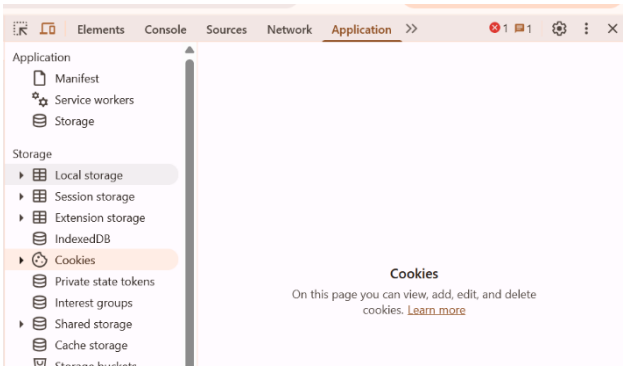
Validate token length and format strictly to prevent overflow or malformed input

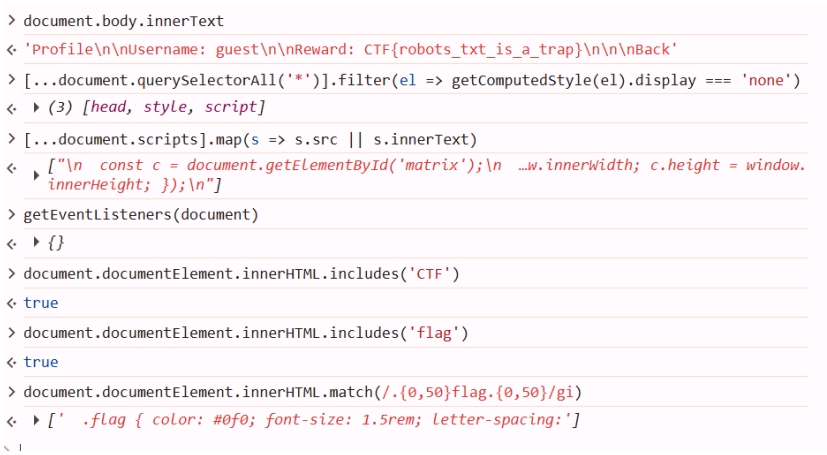
Avoid substring or prefix-based token checks, always compare full values securely

Log and rate-limit failed token attempts to detect brute-force or abuse

Don't rely only on bearer tokens, consider using sessions or multi-factor authentication

This challenge helped me practice how to explore a web app with minimal information, follow hints, and test for logic bugs in token validation. Even though I didn't solve it, I documented everything I tried and learned, which is still valuable for understanding how these vulnerabilities work. I think that I was overthinking a bit, since the hometasks went smoother, but I had no Idea where the flag could be.





- **Console Inspection**

  To check for hidden Clues by the end, I opened the browser console and ran several JavaScript commands. I used `document.body.innerText` to reveal all visible text, which included the fake flag: `CTF{robots_txt_is_a_trap}`. I confirmed that no other elements were hidden using `getComputedStyle`, and verified that the page contained the word “flag” with `document.documentElement.innerHTML.includes('flag')`. I also ran a regex match to locate the word “flag” in the HTML, which pointed to a CSS class definition (`.flag { … }`), not an actual flag value. These checks confirmed that the flag was embedded in visible content and not hidden in scripts or the DOM.