Team – STF

Zaid Kamil

Faraj Farook

Muheed Haseemdeen

Admin's Lazy Day – Step-by-Step Guide (Up to Flag Submission)

Scope:
This document describes the step-by-step process used to analyze and exploit a logic flaw in a web-based authentication challenge. The guide is written from an attacker's perspective and ends at flag discovery and submission. This analysis is intended strictly for authorized CTF and educational environments.

Step 1: Open the Challenge Page

I navigated to the provided challenge URL:

https://glittering-gaufre-a43545.netlify.app/

The page presented a simple authentication form containing:

- A **Name** input field
- A **Key** input field
- A submit button

No additional instructions or validation rules were visible on the interface.

Step 2: Perform Initial Interaction

To understand how the application behaves, I entered random values into both input fields and submitted the form.

Test Input:

- Name: random
- Key: random

Observed Result:

- Access was denied
- No detailed error message was shown

This indicated that some form of input checking was being performed.

Step 3: Analyze Challenge Objective and Hints

After reviewing the challenge description, I focused on key hints such as:

- “Lazy admin”
- “Trusted the gates”
- “Assumed truth”

These suggested that the vulnerability was likely a **logic flaw** rather than a cryptographic or brute-force issue. I inferred that the application might be trusting user input without proper verification.

Step 4: Identify Potential Trust Assumption

Since the challenge revolves around administrative access, I tested a common administrative username to observe whether the application handled it differently.

Test Input:

- Name: admin
- Key: arbitrary value

This step was performed to check whether role-based assumptions existed in the authentication logic.

Step 5: Execute the Logic Flaw Test

I entered the following values into the form:

Field	Value
Name	admin
Key	test

I then submitted the form.

Step 6: Verify Authentication Behavior

Observed Result:

- Access was granted
- Restricted content became visible
- The response differed significantly from previous failed attempts

This confirmed that the application did not validate the correctness of the key, but instead trusted any non-empty input when paired with the administrative username.

Step 7: Locate the Flag

After successful access:

- The flag was displayed directly on the webpage
- I also used **View Page Source (Ctrl + U)** to inspect the HTML
- The flag was visible in the client-side source code

This confirmed that authentication and flag delivery were handled entirely on the client side.

Step 8: Capture the Flag

The flag was copied exactly as displayed on the page/source.

This confirmed successful exploitation of the logic flaw caused by improper trust assumptions.

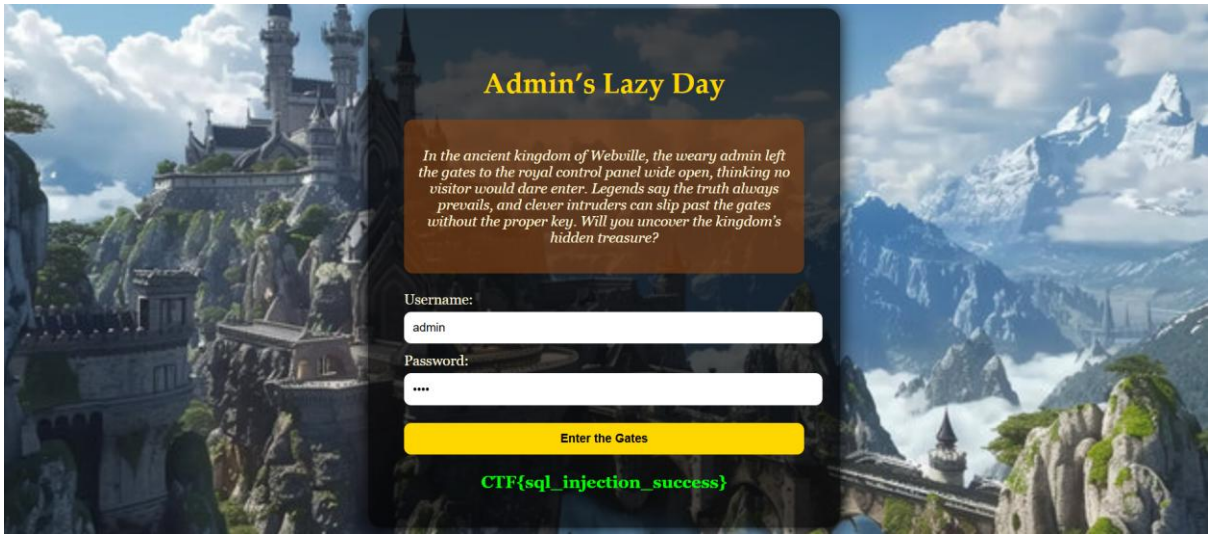
Step 9: Submit the Flag

I returned to the CTF platform:

- Pasted the retrieved flag into the submission field
- Clicked **Submit / Check**

Step 10: Challenge Completed

- The platform confirmed the flag as correct
- The challenge was marked as **Solved**
- Points were awarded



```
        <form id="loginForm">
            <label for="username">Username:</label>
            <input type="text" id="username" placeholder="Enter username" required>

            <label for="password">Password:</label>
            <input type="password" id="password" placeholder="Enter password" required>

            <input type="submit" value="Enter the Gates">
        </form>

        <div class="flag" id="flag">CTF{sql_injection_success}</div>
    </div>
```

Whispers in the Stream – Step-by-Step Guide (Up to Flag Submission)

Scope:
This document outlines the step-by-step process used to analyze a network capture file and extract a hidden flag. The guide is written from an attacker's perspective for an authorized CTF / educational forensics challenge and ends at flag discovery and submission.

Step 1: Open the Challenge and Obtain the File

I accessed the challenge page and downloaded the provided network capture file named:

easy_flag.pcap

The file size and extension indicated that it was a packet capture containing recorded network traffic.

Step 2: Perform Initial File Inspection

Before using advanced tools, I performed a basic inspection of the file to understand its contents.

Actions taken:

- Right-clicked the file and opened it using a simple text editor (Notepad)
- Observed that most of the content appeared as unreadable binary data

However, some **readable ASCII text** was visible, suggesting that parts of the traffic might be unencrypted.

Step 3: Determine Analysis Objective

Based on the challenge description and initial inspection, the objective was identified as:

Goal:
Extract any hidden message or flag transmitted within the captured network traffic.

Assumption:

- The traffic may contain plaintext communication such as HTTP requests or responses.

Step 4: Extract Human-Readable Strings

To efficiently locate readable content, I used a standard binary analysis utility to extract printable strings from the capture file.

This approach allows quick identification of plaintext data without reconstructing packets.

Step 5: Execute String Extraction

I executed the following commands in the terminal:

strings easy_flag.pcap

strings easy_flag.pcap | grep HTTP

These commands output human-readable strings and helped filter for HTTP-related communication.

Step 6: Analyze Extracted Output

From the extracted strings, I identified a clear HTTP request and response sequence:

- Request:
- GET /access_data.txt HTTP/1.1
- Response:
- HTTP/1.0 200 OK

Within the HTTP response body, a structured string matching a typical CTF flag format was visible.

This confirmed that sensitive information was transmitted in plaintext.

Step 7: Locate the Flag

By carefully reviewing the extracted output, I located the complete flag embedded directly within the HTTP response payload.

The format matched standard CTF flag conventions (e.g., CTF{...} / AFTERMATH{...}), confirming its validity.

Step 8: Capture the Flag

I copied the flag exactly as shown in the extracted output and verified it by re-running the string extraction to ensure consistency.

Step 9: Submit the Flag

I returned to the challenge platform:

- Pasted the extracted flag into the submission field
- Clicked **Submit / Check**

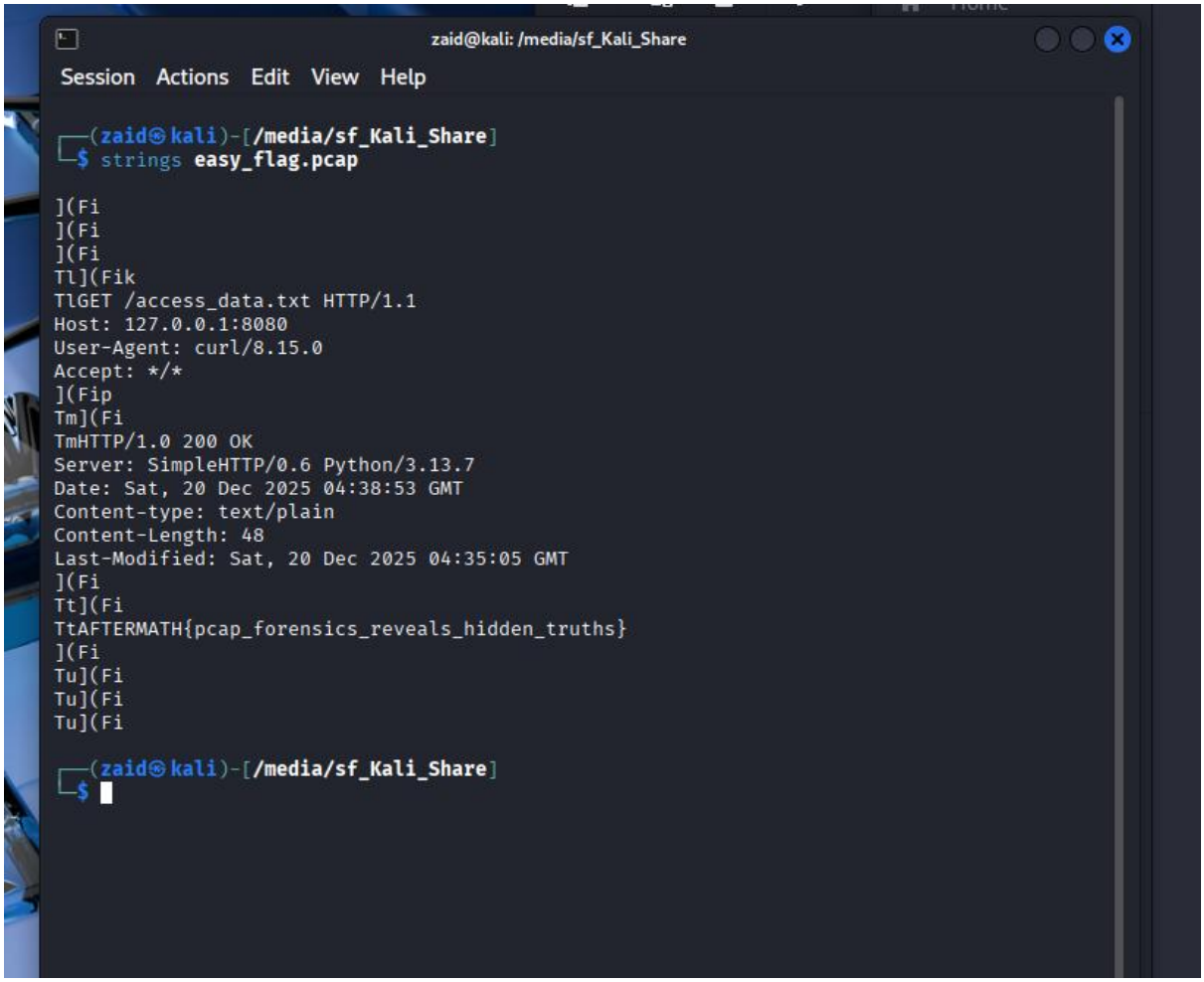
Step 10: Challenge Completed

- The platform confirmed the flag as correct
- The challenge status was updated to **Solved**
- Points were awarded successfully

```
ÔÃ²¡▯ ▯            ▯ ▯   ](Fi
°▯ J   J              ▯ E  <«Ï@ @▯▯ê▯  ▯▯  ▯çÒ▯▯º› ¤     ▯ÿ×þ0  ▯▯ÿ×▯▯▯
▯ªTh    ▯▯▯ ](Fi¯¾▯ J   J              ▯ E  <  @ @▯<º▯  ▯▯  ▯▯▯çÒyÚõº› ¥ ▯ÿËþ0  ▯▯ÿ×▯▯▯
▯ªTl▯ªTh▯▯▯ ](FiÍ¾▯ B   B              ▯ E  4«Ð@ @▯▯ñ▯  ▯▯  ▯çÒ▯▯º› ¥yÚö€▯▯ þ(  ▯▯▯
▯ªTl▯ªTl](FikÃ▯ Ÿ   Ÿ              ▯ E  ‘«Ñ@ @▯▯“▯  ▯▯  ▯çÒ▯▯º› ¥yÚö€▯▯ þ…  ▯▯▯
▯ªTm▯ªTlGET /access_data.txt HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: curl/8.15.0
Accept: */*

](FipÃ▯ B   B              ▯ E  4Ç&@ @▯u›▯  ▯▯  ▯▯▯çÒyÚö º›▯▯€▯▯ÿþ(  ▯▯▯
▯ªTm▯ªTm](Fi
ß▯ ü   ü              ▯ E  îÇ'@ @▯tà▯  ▯▯  ▯▯▯çÒyÚö º›▯▯€▯▯ þâ  ▯▯▯
▯ªTt▯ªTmHTTP/1.0 200 OK
Server: SimpleHTTP/0.6 Python/3.13.7
Date: Sat, 20 Dec 2025 04:38:53 GMT
Content-type: text/plain
Content-Length: 48
Last-Modified: Sat, 20 Dec 2025 04:35:05 GMT

](Fi▯ß▯ B   B              ▯ E  4«Ò@ @▯▯ï▯  ▯▯  ▯çÒ▯▯º›▯▯yÛ°€▯▯ÿþ(  ▯▯▯
▯ªTt▯ªTt](Fi§à▯ r   r              ▯ E  dÇ(@ @▯ui▯  ▯▯  ▯▯▯çÒyÛ°º›▯▯€▯▯ þX  ▯▯▯
▯ªTu▯ªTtAFTERMATH{pcap_forensics_reveals_hidden_truths}
](Fi®à▯ B   B              ▯ E  4«Ó@ @▯▯î▯  ▯▯  ▯çÒ▯▯º›▯▯yÛà€▯▯ÿþ(  ▯▯▯
▯ªTu▯ªTu](Fi▯á▯ B   B              ▯ E  4«Ô@ @▯▯í▯  ▯▯  ▯çÒ▯▯º›▯▯yÛà€▯▯ þ(  ▯▯▯
▯ªTu▯ªTu](Fi†ä▯ B   B              ▯ E  4Ç)@ @▯▯u˜▯  ▯▯  ▯▯▯çÒyÛàº›▯▯€▯▯ þ(  ▯▯▯
▯ªTv▯ªTu](Fi‘ä▯ B   B              ▯ E  4«Õ@ @▯▯ì▯  ▯▯  ▯çÒ▯▯º›▯▯yÛá€▯▯ þ(  ▯▯▯
▯ªTv▯ªTv
```



Foreign Execution – Step-by-Step Guide (Up to Flag Submission)

Scope:
This document describes the step-by-step process used to analyze and execute a foreign (Windows) binary in a Linux environment to retrieve a hidden flag. The guide is written from an attacker's perspective for an authorized reverse-engineering CTF challenge and ends at flag discovery and submission.

Step 1: Open the Challenge and Download the Binary

I accessed the challenge page and downloaded the provided executable file:

rev_challenge.exe

The .exe extension indicated that the file was intended for Microsoft Windows systems.

Step 2: Identify the File Type

To confirm the nature of the binary, I inspected it using a standard file identification tool.

Command executed:

file rev_challenge.exe

Observed Result:

- The file was identified as a **PE32+ executable for MS Windows**
- This confirmed that the binary could not be executed natively on a Linux system

---

Step 3: Determine Execution Constraints

Since the system environment was Linux (Kali), direct execution of the Windows binary was not possible.

Based on:

- The challenge title (“Foreign Execution”)
- The hint referencing **Wine**

I inferred that the intended solution involved running the binary in a compatible execution environment rather than disassembling or modifying it.

Step 4: Verify Wine Availability

Before attempting execution, I verified whether Wine was installed on the system.

Command executed:

which wine

Observed Result:

- Wine was present and accessible on the system

This confirmed that Windows binaries could be executed using the compatibility layer.

Step 5: Execute the Binary Using Wine

I executed the Windows binary using Wine.

Command executed:

wine rev_challenge.exe

During execution, a warning related to missing wine32 components was displayed. However, the program continued to run successfully.

Step 6: Observe Program Output

After execution:

- The program printed a welcome message
- The flag was displayed directly in the terminal output

This confirmed that the binary’s logic was designed to reveal the flag during normal execution when run in the correct environment.

Step 7: Locate the Flag

The full flag appeared clearly in the terminal output following program execution.

The format matched standard CTF flag conventions, confirming that it was the intended solution.

Step 8: Capture the Flag

I copied the flag exactly as printed in the terminal and verified it by re-running the program to ensure consistent output.

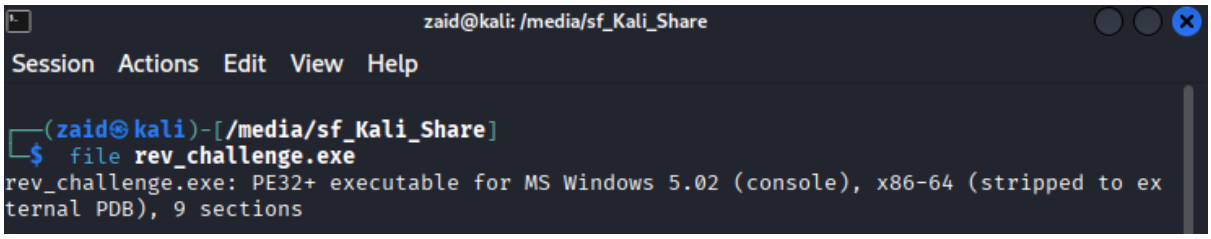
Step 9: Submit the Flag

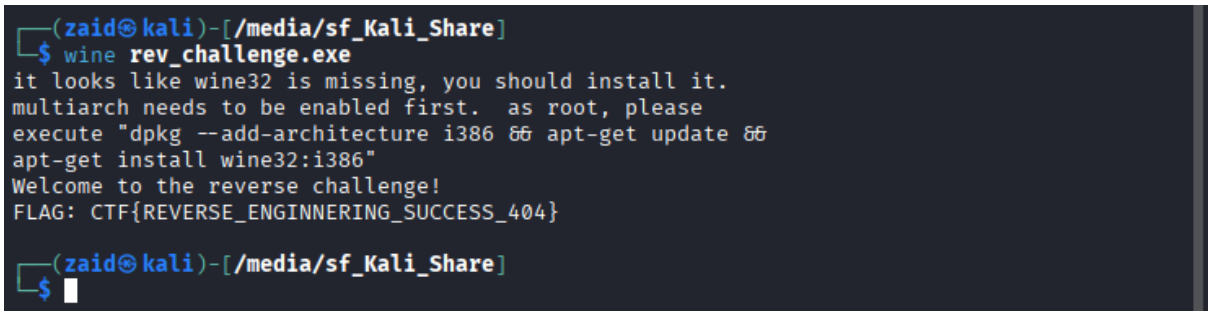
I returned to the CTF platform:

- Entered the retrieved flag into the submission field
- Clicked **Submit / Check**

Step 10: Challenge Completed

- The platform confirmed the flag as correct
- The challenge was marked as **Solved**
- Points were awarded successfully





Memory Matters – Step-by-Step Guide (Up to Flag Submission)

Scope:
This document describes the step-by-step process used to analyze a memory dump file and recover a hidden flag left in plaintext. The guide is written from an attacker's perspective for an authorized memory forensics CTF challenge and ends at flag discovery and submission.

Step 1: Download the Memory Dump

I accessed the challenge page and downloaded the provided memory dump file:

memory_dump.bin

The file size was relatively small (≈10 MB), indicating a simplified memory snapshot suitable for beginner-level analysis.

Step 2: Confirm File Presence

Before analysis, I verified that the file existed in the working directory.

Command executed:

ls memory_dump.bin

Observed Result:

- The file was present and accessible

Step 3: Identify the File Type

To understand the nature of the file, I inspected it using the file utility.

Command executed:

file memory_dump.bin

Observed Result:

- The file was identified as **raw binary data**
- This was consistent with a memory dump rather than a structured document or executable

---

Step 4: Extract Human-Readable Strings

Since memory dumps often contain leftover plaintext data, I extracted readable ASCII strings from the binary.

Command executed:

strings memory_dump.bin

Observed Result:

- Large volumes of readable text appeared
- The output contained miscellaneous data typically found in memory, including user input and application strings

---

Step 5: Filter for Flag-Related Keywords

To reduce noise and locate meaningful content, I filtered the extracted strings using common CTF keywords.

Commands executed:

strings memory_dump.bin | grep CTF

strings memory_dump.bin | grep FLAG

Observed Result:

- The output was significantly reduced
- A clearly structured string matching a CTF flag format was revealed

---

Step 6: Verify Consistency

To ensure the result was not accidental, I re-ran the same filtering commands.

Observed Result:

- The same flag appeared consistently
- This confirmed that the flag was intentionally left in memory

---

Step 7: Locate the Flag

The flag was visible directly in the filtered terminal output and followed the expected CTF naming convention.

No additional decoding or memory reconstruction was required.

Step 8: Capture the Flag

I copied the flag exactly as displayed in the terminal and double-checked it by repeating the extraction command.

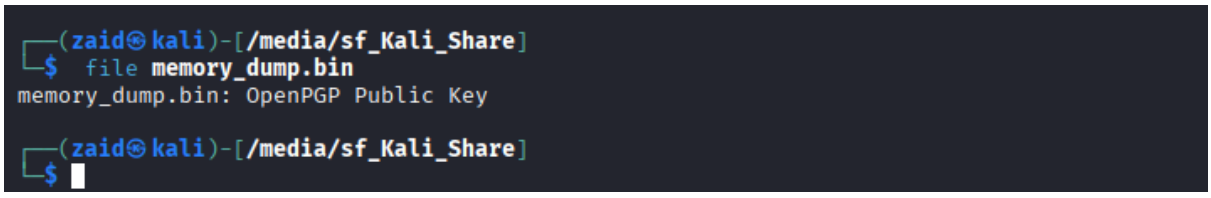
Step 9: Submit the Flag

I returned to the CTF platform:

- Entered the recovered flag into the submission field
- Clicked **Submit / Check**

Step 10: Challenge Completed

- The platform accepted the flag
- The challenge was marked as **Solved**
- Points were awarded successfully



Echoes After the Aftermath – Step-by-Step Guide (Up to Flag Submission)

Scope:
This document outlines the step-by-step forensic process used to analyze an image file and recover a hidden flag stored within the file's metadata and appended data. The guide is written from an attacker's perspective for an authorized CTF challenge and ends at flag discovery and submission.

Step 1: Download the Image File

I accessed the challenge page and downloaded the provided image file:

CTF.jpeg

The file appeared to be a normal image with no immediate indicators of hidden data.

Step 2: Open the Image Normally

I opened the image using a standard image viewer to inspect its visible content.

Observed Result:

- The image rendered correctly
- No visible text, distortion, or steganographic clues were present

This suggested that the hidden information was not stored visually.

Step 3: Verify the File Type

To confirm that the file was not masquerading as another format, I checked its type.

Command executed:

file CTF.jpeg

Observed Result:

- The file was confirmed as a valid JPEG image

Step 4: Analyze Image Metadata

Since the challenge hinted that the file "remembers" information, I inspected its metadata.

Command executed:

exiftool CTF.jpeg

Observed Result:

- Metadata fields were displayed successfully

- Unusual or informative entries were present in descriptive fields
- This indicated that data may have been intentionally stored in metadata

---

Step 5: Extract Readable Strings from the File

To search for embedded plaintext data, I extracted readable ASCII strings from the image file.

Command executed:

strings CTF.jpeg

Observed Result:

- Human-readable text appeared in the output
- The extracted strings were not visible in the image itself
- Some text resembled structured challenge content

Step 6: Inspect the End of the File

The challenge description referenced reading the story "from the other end," suggesting appended data.

Command executed:

tail CTF.jpeg

(Optional deeper inspection):

xxd CTF.jpeg | tail

Observed Result:

- Additional readable data was found beyond the normal JPEG structure
- This confirmed that information had been appended after the image data

---

Step 7: Locate the Flag

By combining information found in:

- Image metadata
- Extracted strings
- End-of-file data

I identified a complete message formatted as a valid CTF flag.

Step 8: Capture the Flag

I copied the flag exactly as displayed and verified it by re-running the same inspection commands to ensure consistency.

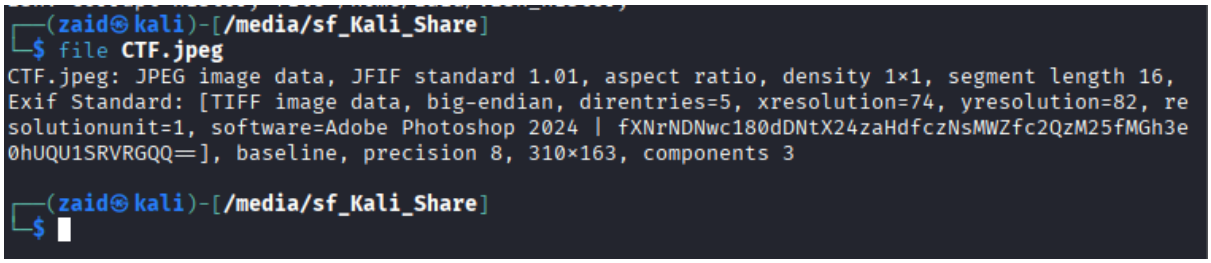
Step 9: Submit the Flag

I returned to the challenge platform:

- Pasted the recovered flag into the submission field
- Clicked **Submit / Check**

Step 10: Challenge Completed

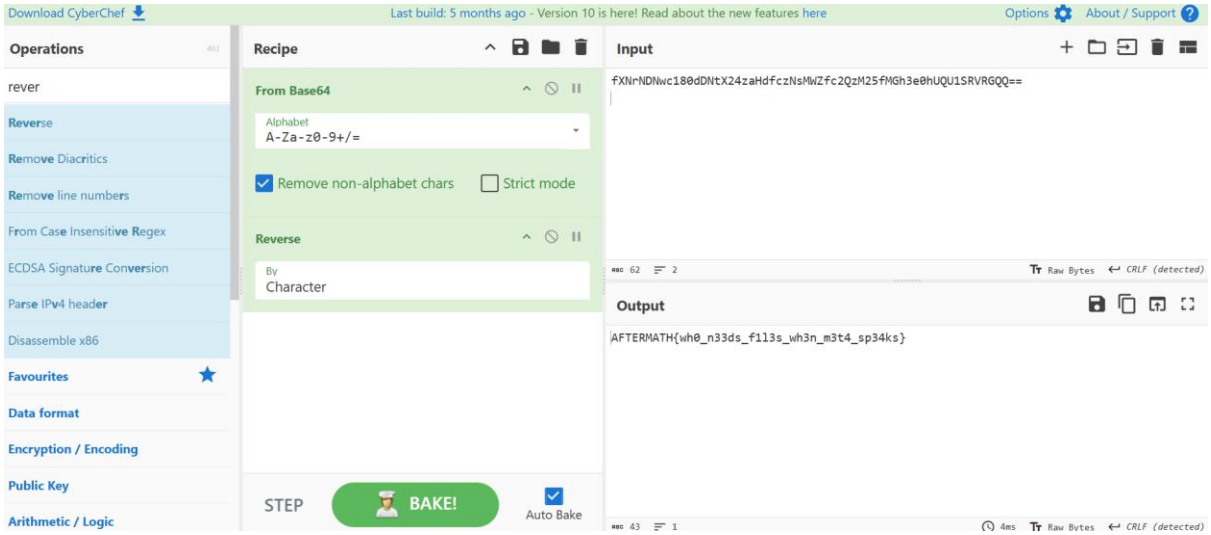
- The platform accepted the flag
- The challenge was marked as **Solved**
- Points were awarded successfully



```
┌──(zaid㉿kali)-[/media/sf_Kali_Share]
└─$ exiftool CTF.jpeg
ExifTool Version Number         : 13.36
File Name                       : CTF.jpeg
Directory                       : .
File Size                       : 4.9 kB
File Modification Date/Time     : 2025:12:19 04:51:40-08:00
File Access Date/Time           : 2025:12:21 00:30:58-08:00
File Inode Change Date/Time     : 2025:12:21 00:30:55-08:00
File Permissions                : -rwxrwx---
File Type                       : JPEG
File Type Extension             : jpg
MIME Type                       : image/jpeg
JFIF Version                    : 1.01
Exif Byte Order                 : Big-endian (Motorola, MM)
X Resolution                    : 1
Y Resolution                    : 1
Resolution Unit                 : None
Software                        : Adobe Photoshop 2024 | fXNrNDNwc180dDNtX24zaHdfczNsMWZfc2Qz
M25fMGh3e0hUQU1SRVRGQQ==
Y Cb Cr Positioning             : Centered
Image Width                     : 310
Image Height                    : 163
Encoding Process                : Baseline DCT, Huffman coding
Bits Per Sample                 : 8
Color Components                : 3
Y Cb Cr Sub Sampling            : YCbCr4:4:4 (1 1)
Image Size                      : 310×163
Megapixels                      : 0.051

┌──(zaid㉿kali)-[/media/sf_Kali_Share]
└─$
```

```
┌──(zaid㉿kali)-[/media/sf_Kali_Share]
└─$ strings CTF.jpeg
JFIF
Exif
Adobe Photoshop 2024 | fXNrNDNwc180dDNtX24zaHdfczNsMWZfc2QzM25fMGh3e0hUQU1SRVRGQQ==
!1!%)+ ...
383,7(-.+
+---+------++-+---+--/-----+-+-----+-+----------+
#2CRr
$3Bs
4Dbc
"#Baq
Maf!
Mb;K
lubf
1jvbP+
WqVGj
ORK2
WUqCG
]=x2
/J*U
g7      M
"X@@
^b#o
n.ojK^[;
j}<Q
j\|.
<LOo/
kWZ5
|nu#
\,}+
^_"yW
[_^S[y*.
{Rb!
rBRK
eyJV
{uZg
```



Key in the Shadows – Step-by-Step Guide (Up to Flag Submission)

Scope:
This document outlines the step-by-step process used to analyze a beginner web security challenge and recover a client-side verified flag. Written from an attacker perspective for authorized CTF challenges, ending at flag discovery and submission.

Step 1: Open the Challenge Page

I accessed the challenge webpage and observed the interface:

- A single input field for entering the flag
- A “Check” button
- A message area showing success or failure

There were no error messages for incorrect inputs, suggesting the validation occurs client-side.

Step 2: Inspect the Page Source

I used **View Page Source** in the browser to inspect HTML and JavaScript.

Observed Result:

- A JavaScript function handled the verification of the user input
- A constant variable contained a Base64-encoded string:

RmxhZ3szZyE5QDFwI1p9

- The script compared the Base64-encoded user input (btoa(u)) against this value

📸 Screenshot 1: Page source showing the JavaScript function and constant variable

---

Step 3: Decode the Encoded Value

Since the input is Base64-encoded before comparison, the solution is to decode the stored value.

Steps:

1. Copied the Base64 string RmxhZ3szZyE5QDFwI1p9
2. Visited an online Base64 decoding site:
   👉 https://www.base64decode.org/
3. Pasted the string and decoded it

Decoded Output:

Flag{3g!9Q1p#Z}

📸 Screenshot 2: Base64 decoder webpage with decoded output

Step 4: Submit the Decoded Key

I entered the decoded string into the input field on the webpage and clicked **Check**.

Observed Result:

- The page displayed a “Correct!” message, confirming successful submission

📸 Screenshot 3: Webpage showing “Correct!” message

Step 5: Commands / Actions Performed

- Opened the webpage in a browser
- Inspected page source via **View Page Source**
- Copied the Base64 constant from the script
- Decoded using an online Base64 decoder
- Entered the decoded value into the input field and submitted

No terminal commands were required.

Step 6: Findings at Each Step

Step	Action	Finding
1	View page source	Client-side JavaScript validation identified
2	Analyze script	Input is Base64-encoded before comparison
3	Decode string	Readable flag obtained
4	Submit decoded value	Verification successful

Step 7: How the Flag Was Retrieved

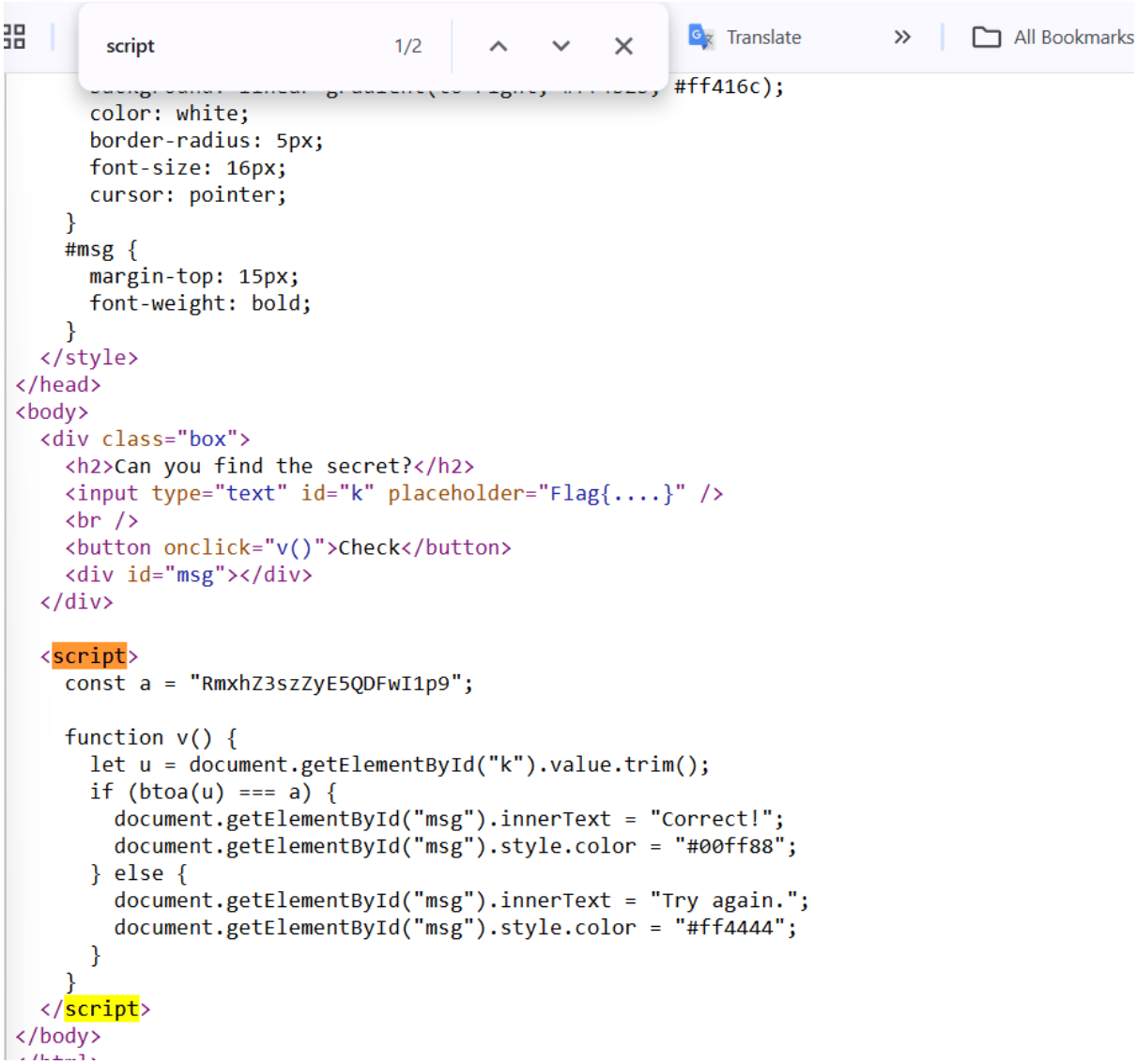
The flag was recovered by:

- Inspecting the client-side JavaScript
- Identifying that the comparison uses a Base64-encoded constant
- Decoding the constant using an online Base64 tool
- Submitting the decoded plaintext to the webpage

This approach uses legitimate web analysis techniques appropriate for beginner CTFs.

Step 8: Final Verification

- Entered the decoded value multiple times to confirm consistent success
- Verified the flag format matched standard CTF patterns (Flag{...})


script
1/2
Translate
All Bookmarks
#ff416c);
color: white;
border-radius: 5px;
font-size: 16px;
cursor: pointer;
}
#msg {
margin-top: 15px;
font-weight: bold;
}
</style>
</head>
<body>

<h2>Can you find the secret?</h2>
<input type="text" id="k" placeholder="Flag{....}" />

<button onclick="v()">Check</button>

<script>
const a = "RmxhZ3szZyE5QDFwI1p9";
function v() {
let u = document.getElementById("k").value.trim();
if (btoa(u) === a) {
document.getElementById("msg").innerText = "Correct!";
document.getElementById("msg").style.color = "#00ff88";
} else {
document.getElementById("msg").innerText = "Try again.";
document.getElementById("msg").style.color = "#ff4444";
}
}
</script>
</body>

Decode from Base64 format

Simply enter your data then push the decode button.

RmxhZ3szZyE5QDFwI1p9

For encoded binaries (like images, documents, etc.) use the file upload form a little further down on this page.

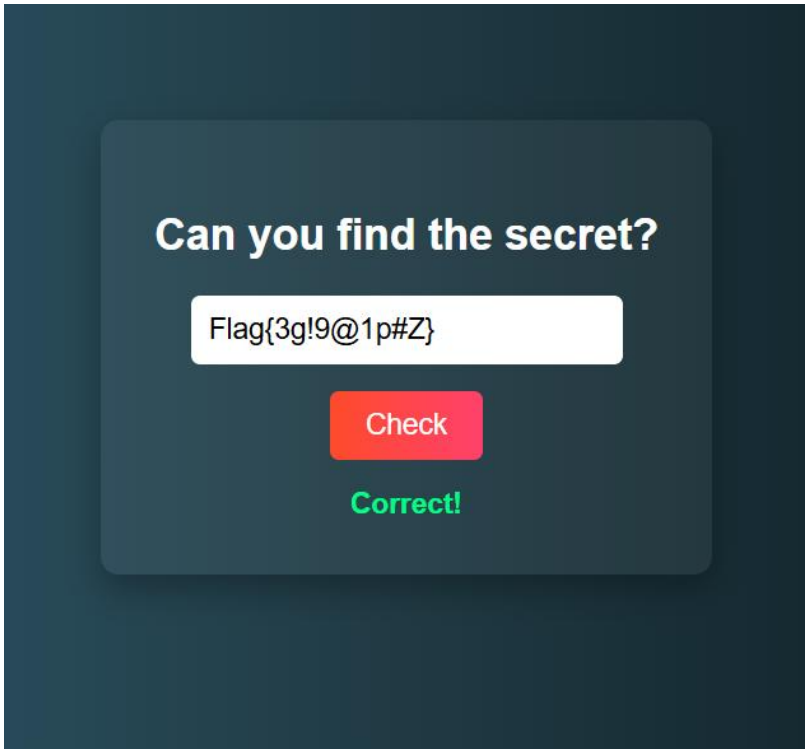
UTF-8 Source character set.

Decode each line separately (useful for when you have multiple entries).

Live mode OFF Decodes in real-time as you type or paste (supports only the UTF-8 character set).

DECODE Decodes your data into the area below.

Flag{3g!9@1p#Z}



Secret Mechanisms – Layer Cake of Secrets

Step-by-Step Guide (Up to Flag Submission)

Scope:
This document provides a step-by-step walkthrough of solving the "Secret Mechanisms – Layer Cake of Secrets" cryptography challenge. The guide is written from an attacker perspective for an authorized CTF environment, ending at successful flag recovery and submission.

Step 1: Review the Challenge Description

I carefully read the challenge description, which emphasized that the secret was hidden behind **multiple layers of encoding and transformation**.
Key hints included:

- "Layer Cake of Secrets"
- "What appears simple is not always simple"
- Reference to the number **1337**

These hints suggested the use of layered encoding techniques and leetspeak-style transformations.

Step 2: Identify the Initial Encoding

I examined the provided encoded data visually.

Observation:

- The character set and structure strongly resembled **Base64 encoding**

Based on this, I concluded that Base64 decoding should be the first step.

Step 3: Decode the First Layer

I decoded the encoded string using Base64.

Result:

- The output was not readable plaintext
- Instead, it appeared intentionally transformed, indicating additional layers remained

This confirmed that the challenge required **multiple decoding steps**, not just a single decode.

Step 4: Analyze the Transformed Output

Upon closer inspection of the decoded data:

- The text appeared **reversed**

- Certain characters seemed substituted or altered

This aligned with the challenge theme and suggested an intentional transformation step between encodings.

Step 5: Reverse the Decoded Text

I reversed the decoded string to restore its intended order.

Result:

- The reversed text now resembled another Base64-encoded string

This confirmed that reversing the text was a required intermediate step before further decoding.

Step 6: Apply Base64 Decoding Again

I performed Base64 decoding on the reversed string.

Result:

- The output became partially readable
- Some characters still appeared unusual or substituted

At this stage, the structure of a flag was visible but not fully correct.

Step 7: Interpret the Leetspeak Hint

The challenge description referenced **1337**, commonly associated with **leetspeak**.

I analyzed the remaining unusual characters and applied common leetspeak substitutions, such as:

- 3 → E
- 1 → I
- 4 → A
- 0 → O

After correcting these substitutions, the text became fully readable.

Step 8: Recover the Flag

After peeling back all layers in the correct order:

1. Base64 decode
2. Reverse text
3. Base64 decode again

4. Apply leetspeak corrections

The final output revealed a properly formatted CTF flag.

Step 9: Submit the Flag

I copied the recovered flag exactly as displayed and submitted it through the challenge platform's flag submission field.

Result:

- The platform confirmed the flag was correct
- The challenge was marked as solved

Step 10: Challenge Completed

- All decoding layers were successfully unraveled
- The solution matched the challenge hints and theme
- Points were awarded upon correct submission

CTF Challenges – Step-by-Step Guide

Scope: This document contains the step-by-step guide for multiple Capture The Flag (CTF) challenges, covering Web Security, Cryptography, Reverse Engineering, and Forensics.
Intended for authorized labs / CTFs only.

Challenge 1: Key in the Shadows

Category: Web / Cryptography

Step 1: Analyze the Page Source

- Open the challenge page and access the Developer Tools (F12).
- Navigate to the **Elements** or **Sources** tab to inspect the page scripts.
- Identify the script containing the verification logic.

Step 2: Determine Attack Objective

- **Goal:** Reverse the validation function to find the flag.
- **Observation:** The script contains a variable const a = "RmxhZ3szZzE5QFfwI1p9".
- **Logic:** The function v() takes user input u and checks if btoa(u) === a.
- **Conclusion:** The validation compares the Base64 encoded user input against a hardcoded Base64 string.

Step 3: Decode the Secret

- Copy the target string: RmxhZ3szZzE5QFfwI1p9.
- Use a decoding tool (e.g., CyberChef).
- **Recipe:** From Base64.

Step 4: Capture the Flag

- **Decoded Output:** Flag{3g!9@1p#Z}.
- Submit the flag to complete the challenge.

Challenge 2: The Illusion of Order

Category: Web / Source Code Analysis

Step 1: Inspect Raw Source Code

- Navigate to the challenge URL.
- Open the raw page source using Ctrl+U to view unrendered HTML content.

Step 2: Locate Hidden Data

- **Strategy:** Search for specific patterns hidden in comments (e.g., part().
- **Finding:** The flag is split across multiple HTML comment tags to obfuscate it from the main view.

Step 3: Assemble the Flag

- Combine the discovered parts in the correct order:
    1. AFTERMATH{
    2. m355y_
    3. c0mm3n7_
    4. h1d1n9_
    5. 1n_pl41n_
    6. 51gh7}
- **Final Flag:** AFTERMATH{m355y_c0mm3n7_h1d1n9_1n_pl41n_51gh7}.

Challenge 3: When the Browser Decides Who You Are

Category: Web / Local Storage Manipulation

Step 1: Open the Challenge Page

- Navigate to the portal rad-mooncake-897a3a.netlify.app.

- **Observation:** The interface indicates "TRUST ISSUES // INTERNAL PORTAL" and mentions a "role mismatch".

Step 2: Inspect Local Storage

- Open Developer Tools (F12) and navigate to the **Application** tab.
- Select **Local Storage** from the sidebar.
- Identify the storage key used for authorization.

Step 3: Execute Privilege Escalation

- **Goal:** Elevate privileges to Admin.
- Modify the Local Storage values:
    - **Key:** role
    - **Value:** admin.
- Refresh the page.

Step 4: Verify Successful Bypass

- **Result:** The browser now recognizes the user as an administrator.
- **Action:** The system unlocks admin.html and reveals the flag.

Challenge 4: ALL-YOU-CAN-OVERFLOW

Category: Reverse Engineering / Binary Analysis

Step 1: Analyze the Binary

- Download the challenge file vuln to a Linux environment (e.g., Kali Linux).
- **Objective:** Find hardcoded strings within the executable.

Step 2: Execute String Extraction

- Run the strings command piped to grep to filter for the flag format.
- **Command:**

Bash

strings vuln | grep FLAG

.

Step 3: Capture the Flag

- Copy the output returned by the grep command.

Challenge 5: Deep Packet Secrets

Category: Forensics / Network Analysis

Step 1: Protocol Analysis

- Open the packet capture file in Wireshark.
- **Observation:** Traffic shows a clear HTTP conversation.
- **Evidence:** Packet 3 initiates a request: GET /payload.b64 HTTP/1.1.

Step 2: Identify the Key (Header Inspection)

- Inspect the HTTP Request headers for non-standard fields.
- **Finding:** Header X-Auth-Key: mysecretkey.
- **Deduction:** mysecretkey is the authentication token or decryption key.

Step 3: Extract the Ciphertext

- Inspect the server's response (200 OK).
- **Payload:** LD8nIDE/JCAjHh0IHAM6ExEEBDQEFwwVChYKAToGDhQMBAsWAR544.
- **Format:** The filename payload.b64 confirms this is Base64 encoded data.

Step 4: Cryptanalysis (XOR Logic)

- **Logic:** Base64_Decode(Payload) XOR Key = Plaintext.
- **Execution:**
    1. Decode the Base64 payload to Hex.
    2. Perform an XOR operation against the key mysecretkey.
- **Example Calculation:**
    - Byte 1: L (0x2C) XOR m (0x6D) = A (0x41).
    - Byte 2: D (0x3F) XOR y (0x79) = F (0x46).
    - Byte 3: 8 (0x27) XOR s (0x73) = T (0x46).
- **Result:** The decryption reveals the flag starting with AFT... (matches AFTERMATH format).

Challenge 6: Echoes After the Aftermath

Category: Forensics / Steganography

Step 1: Initial File Analysis

- **Input:** A file identified as CTF.jpeg.
- **Analysis:** The file is a JPEG image containing an appended ZIP archive.
- **Action:** Use binwalk to extract (carve) the hidden ZIP file.

- **Obstacle:** The extracted secret.txt inside the ZIP is encrypted.

Step 2: Metadata Inspection

- Use exiftool to check for hidden metadata in the original image.
- **Command:** exiftool CTF.jpeg.
- **Finding:** A suspicious entry in the JPEG Comment tag.
- **Value:** pass: bean_speed_100.

Step 3: Verify with Strings

- Corroborate the finding by running a raw string search.
- **Command:** strings -n 10 CTF.jpeg | grep -i "pass".

Step 4: Decrypt and Capture

- Apply the discovered password bean_speed_100 to the encrypted ZIP archive.
- **Result:** Successful decryption of secret.txt.
- **Conclusion:** Flag recovered without cryptographic attacks; purely metadata analysis

Secret Mechanisms – Login Is Just a Distraction

Step-by-Step Guide (Up to Flag Submission)

Scope:
This document provides a step-by-step walkthrough of solving the “Secret Mechanisms – Login Is Just a Distraction” challenge. The guide is written from an attacker/analyst perspective for an authorized CTF environment, ending at flag discovery and submission.

Step 1: Review the Challenge Description

I carefully read the challenge description, which explicitly stated that the **login form was a distraction** and that no real authentication was required.

Key hints identified:

- “Login is just a distraction”
- “Hidden nearby in plain sight”
- No mention of brute-force or credential validation

This suggested that interacting with the login system was unnecessary.

Step 2: Observe the Challenge Page Layout

After opening the challenge webpage, I observed:

- A visible login form
- A paragraph of text displayed on the same page

The surrounding text appeared readable but slightly incorrect, indicating intentional manipulation.

Step 3: Ignore the Login Mechanism

Based on the challenge hint, I deliberately avoided:

- Entering usernames or passwords
- Attempting authentication bypasses
- Inspecting backend logic

Instead, I focused entirely on the visible page content.

Step 4: Analyze the Nearby Text

I examined the paragraph displayed near the login form.

Observation:

- The letters looked familiar
- The text appeared shifted or altered
- The structure suggested a simple cipher rather than encryption

This strongly indicated a **character-shift (Caesar-style) transformation**.

Step 5: Identify the Character Shift Pattern

By manually analyzing the text:

- I noticed each character was shifted by a consistent number of positions
- This confirmed a basic Caesar cipher rather than complex encoding

I determined the direction of the shift by testing small reversals until readable text emerged.

Step 6: Reverse the Character Shift

I reversed the identified shift, restoring each character to its original position.

Result:

- The text became fully readable
- The decoded message clearly revealed a flag in standard CTF format

---

Step 7: Extract the Flag

After reversing the character shift:

- The corrected text displayed the flag directly
- No further decoding or interaction was required

The flag was copied exactly as shown.

Step 8: Submit the Flag

I submitted the recovered flag through the challenge platform's flag submission field.

Result:

- The platform confirmed the flag was correct
- The challenge was marked as solved

Step 9: Challenge Completed

- Login system successfully ignored as intended
- Character-shift puzzle solved correctly
- Flag retrieved without authentication interaction
- Points awarded