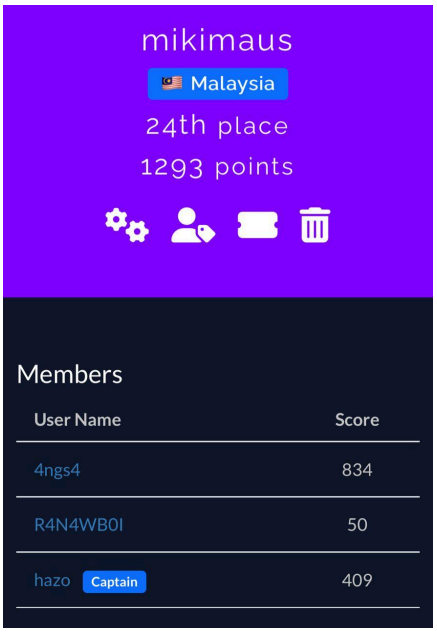
# WARGAMES MY 2024 mikimaus' writeup


Writeup made by:
4ngs4
hazo
R4N4WB0I


*We placed 24th on wgmy24!*

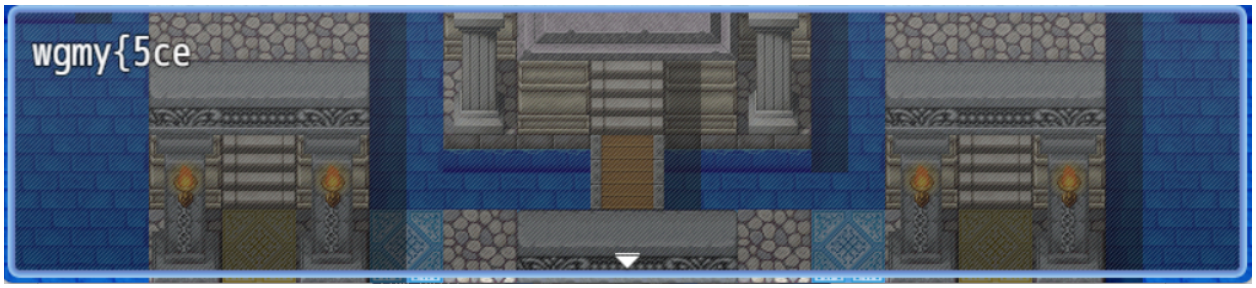
# Table of Content

# GAME SOLVED (1/3)

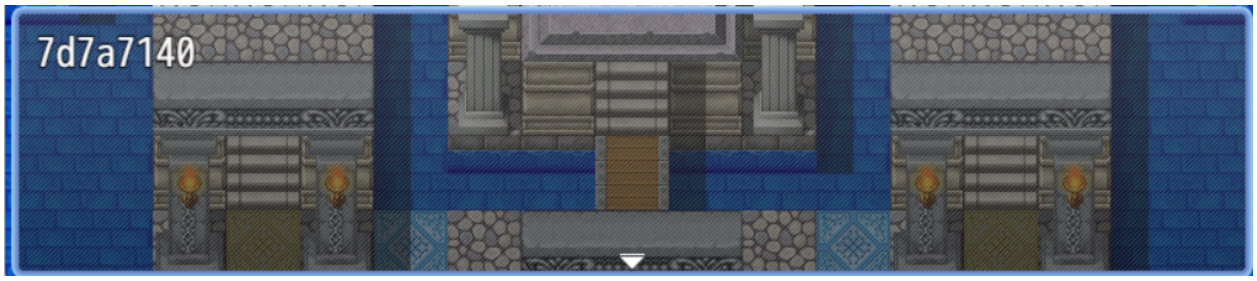
1. World 1


First we were given a World1.exe file and by looking at the interface and template, we could conclude that it is a game made with rpg maker mz.

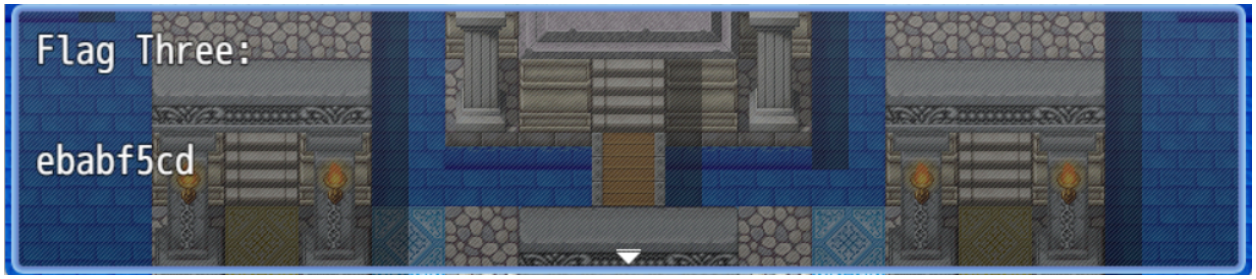

Just like the classic rpg games, we have to save princesses and beat these bosses so I just walk through all the bosses and gain flags.




Flag 1

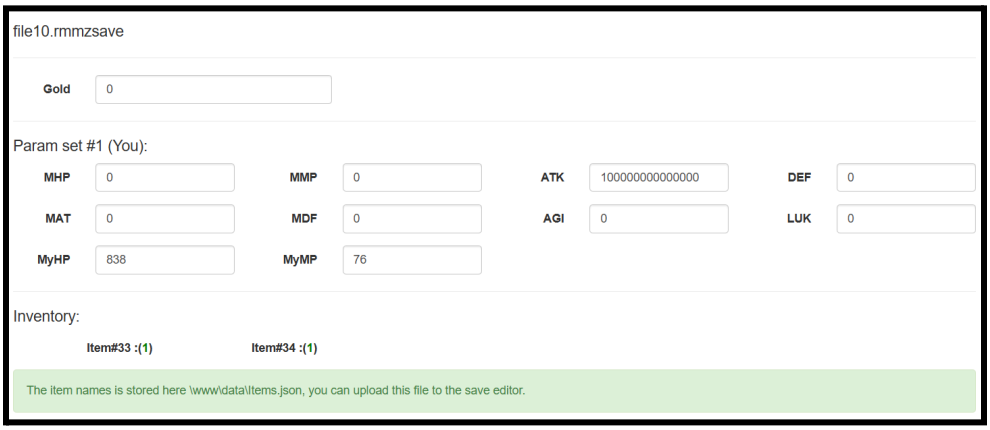



Flag 2




Flag 3


Everything was a smooth sail until I met this absolutely-not-overpowered-boss named Antares (his skill literally dealt 84,200,353). Thus I tried to fight him over and over again and hopefully I would land a critical strike.




On my 20th attempt (yes, I counted), I felt frustrated and realized that I could temper the save file with this sweet save editor I found online.


Then I tried fighting the boss again with my pure skill and technique (hacking technique). I managed to drop an absolutely reasonable skill. Trust me it is super satisfying dealing this much damage against someone who one-shotted you for 20 times.


Then this NPC with an office suit in a castle appeared out of nowhere and told me that the princess was in another castle. Thankfully he gave me a “code”. Since i realized w is the 23th letter in alphabet i knew the password is “wgmy”

I was given a QR code. God forbid these authors to ease our life (jk i love the authors). The flag obtained from the QR is “3fcaac2}”.

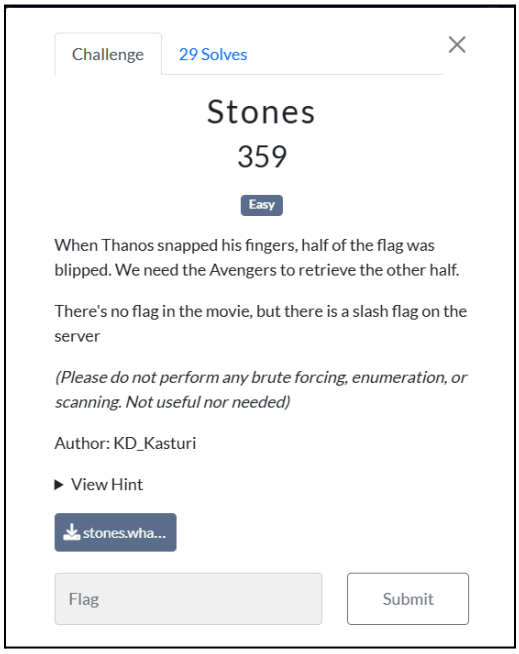

Then I realized there are only 4 flags which means I missed the 4th flag.


After 15 minutes of exploring the map from the start, I finally found the 4th flag on the volcano map.

| Flag: wgmy{5ce7d7a7140ebabf5cd43effd3fcaac2} |
| --- |

# Reverse Solved(1/4)

1. Stones



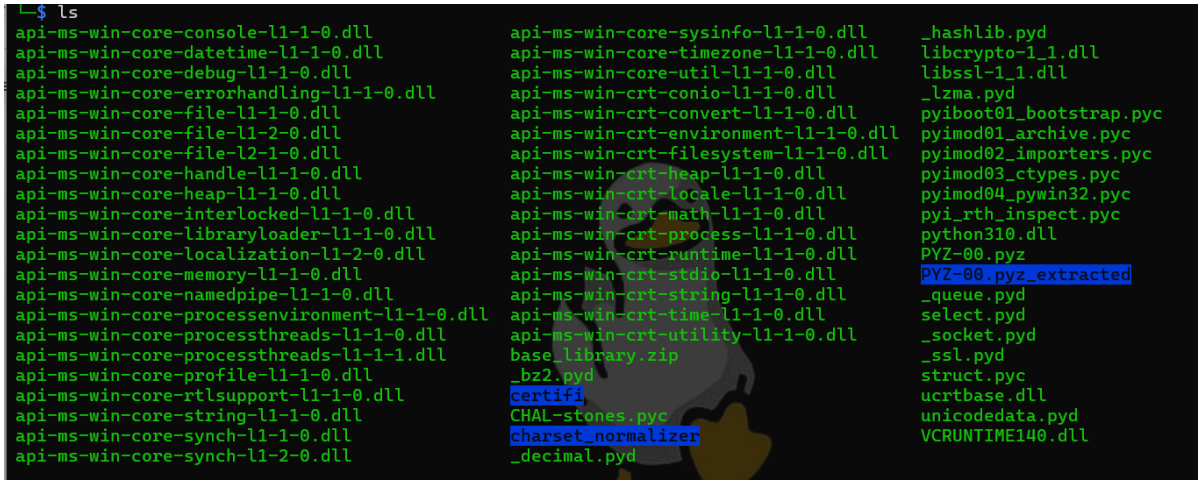

We were provided with a “stones.whatdis” i wonder what file is it

```
└─$ file stones.whatdis
stones.whatdis: PE32+ executable (console) x86-64, for MS Windows, 6 sections
```

By using file command, we can see that it is an exe file so I just rename it to .exe.

```
└─$ ./stones.exe
We're gonna need a really big brain; bigger than his?
Traceback (most recent call last):
  File "urllib3\connection.py", line 198, in _new_conn
  File "urllib3\util\connection.py", line 85, in create_connection
  File "urllib3\util\connection.py", line 73, in create_connection
TimeoutError: [WinError 10060] A connection attempt failed because the connected party did not properly respond after a
period of time, or established connection failed because connected host has failed to respond

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "urllib3\connectionpool.py", line 787, in urlopen
  File "urllib3\connectionpool.py", line 493, in _make_request
  File "urllib3\connection.py", line 445, in request
  File "http\client.py", line 1278, in endheaders
  File "http\client.py", line 1038, in _send_output
  File "http\client.py", line 976, in send
  File "urllib3\connection.py", line 276, in connect
  File "urllib3\connection.py", line 207, in _new_conn
urllib3.exceptions.ConnectTimeoutError: (<urllib3.connection.HTTPConnection object at 0x000001E486AE49D0>, 'Connection t
o 3.142.133.106 timed out. (connect timeout=None)')
```

I tried to run it but seems to be there is an error with the file and it is from a .py file so now we knew that this exe is created using python




Extract the python file from exe using pyinstxtractor and this is the result. We can see there are a lot of files but there is an eye-catching file named “CHAL-stones.pyc”

```
$ cat beep.py
# Source Generated with Decompyle++
# File: CHAL-stones.pyc (Python 3.10)

import requests
from datetime import datetime
from urllib.request import urlopen
from datetime import datetime
server_url = 'http://3.142.133.106:8000/'
current_time = urlopen('http://just-the-time.appspot.com/')
current_time = current_time.read().strip()
current_time = current_time.decode('utf-8')
current_date = current_time.split(' ')[0]
local_date = datetime.now().strftime('%Y-%m-%d')
if current_date == local_date:
    print("We're gonna need a really big brain; bigger than his?")
first_flag = 'WGMY{1d2993'
user_date = current_date
params = {
    'first_flag': first_flag,
    'date': user_date }
response = requests.get(server_url, params, **('params',))
if response.status_code == 200:
    print(response.json()['flag'])
    return None
None(response.json()['error'])
```

By using pycdc, we can decompile the “CHAL-stones.pyc” file and read the content. I threw the code into chatgpt to explain it for me

The program fetches the current date from an external service.
It compares the external date with the local system date.
If they match, it sends a GET request to a server at `http://3.142.133.106:8000/`, including the date and a partial flag (`WGMY{1d2993}`) as parameters.
The server is expected to respond with a full flag, which is printed out.

So I tried to send a request to the url by using curl and received a youtube video which was uploaded on 25th July 2022!

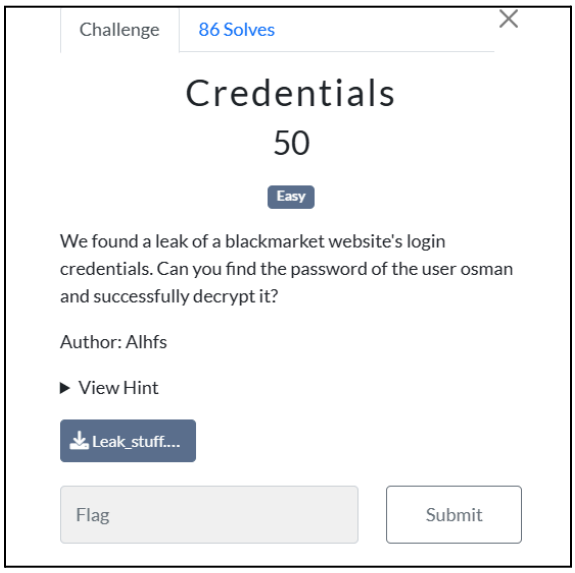
```python
import requests

# Define the server URL and the parameters
server_url = 'http://3.142.133.106:8000/'
params = {
    'first_flag': 'WGMY{1d2993',
    'date': '2022-07-25'
}

# Send the GET request to the server
response = requests.get(server_url, params=params)

# Check if the response status is OK (200)
if response.status_code == 200:
    # If the response contains the flag, print it
    print("Flag received:", response.json()['flag'])
else:
    # If there was an error, print the error message
    print("Error:", response.json().get('error', 'Unknown error'))
```

As usual, I asked chatgpt politely to modify the program and put the right date to get the flag.

Flag: wgmy{1d2993fc6327746830cd374debcb98f5}

# Crypto Solved (1/5)

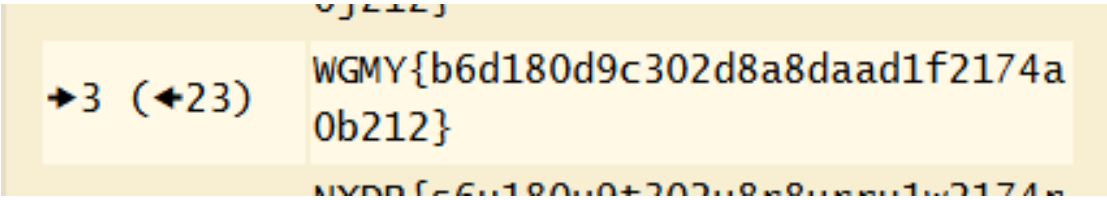
1. Credentials




At first, we were provided with user.txt and passwd.txt. We were asked to find Osman's Password.


We need to find Osman's password so first we found out that Osman's name is on line 337 thus the password must be on line 337 too.


On line 337 in passwd.txt, we could find the ZJPB{e6g180g9f302g8d8gddg1i2174d0e212}. Looks like a flag format!

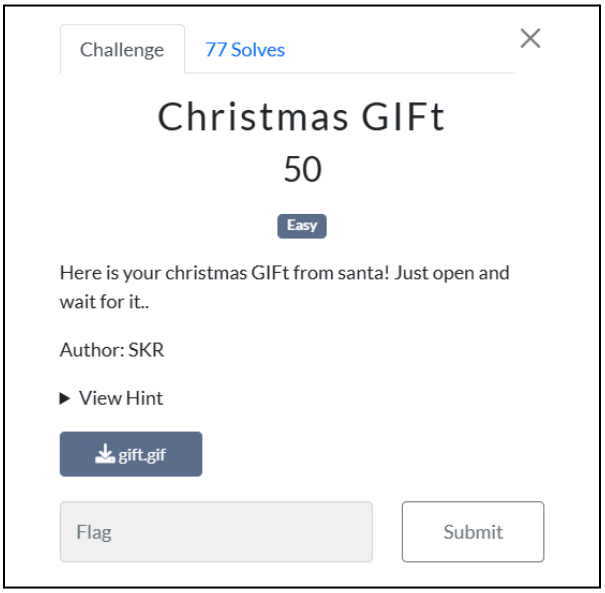



Since ZJPB = wgmy, we can simply conclude that it is a caesar cipher because it uses the substitution method (z=w and j=g is both 3 steps back). Just put it on decode and bruteforce the output.

| Flag: wgmy{b6d180d9c302d8a8daad1f2174a0b212} |
| --- |

# MISC Solved(3/4)

1. Christmas GIFt




A Christmas gift! An interesting challenge indeed!

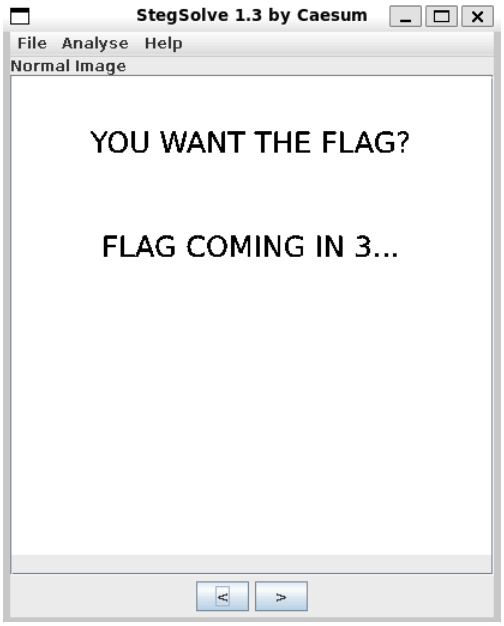

Inspecting the gif, we could guess that this gif exposed something at the end but it is too fast for my weak eyes.. So i decided to use stegsolve to inspect frame by frame


So there are 1402 frames and the flag is in the last frame!

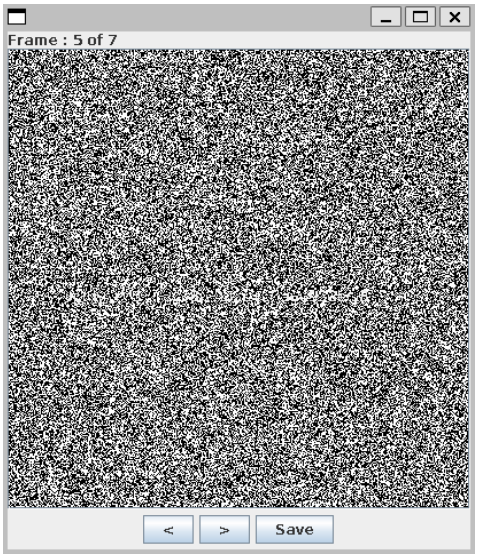
Flag: wgmy{1eaa6da7b7f5df6f7c0381c8f23af4d3}

2. Invisible ink

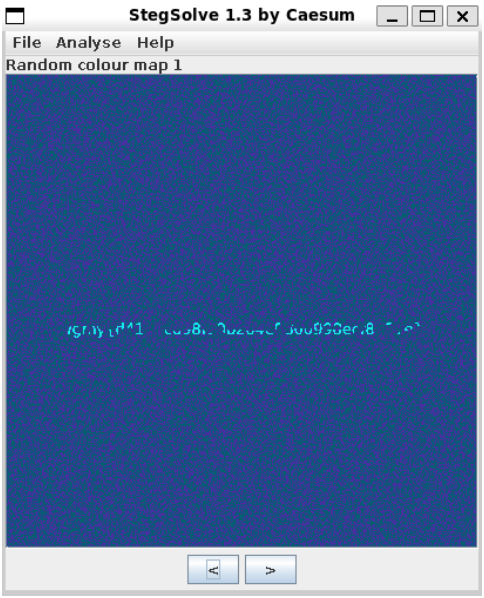

We were provided with another interesting gif!




I decided to use stegsolve again to inspect the gif since aperisolve and ezgif didnt help much




By using the frame browser, we could inspect that there are weird images on frame 5 and 6! Save the frames as .bmp (bitmap file)

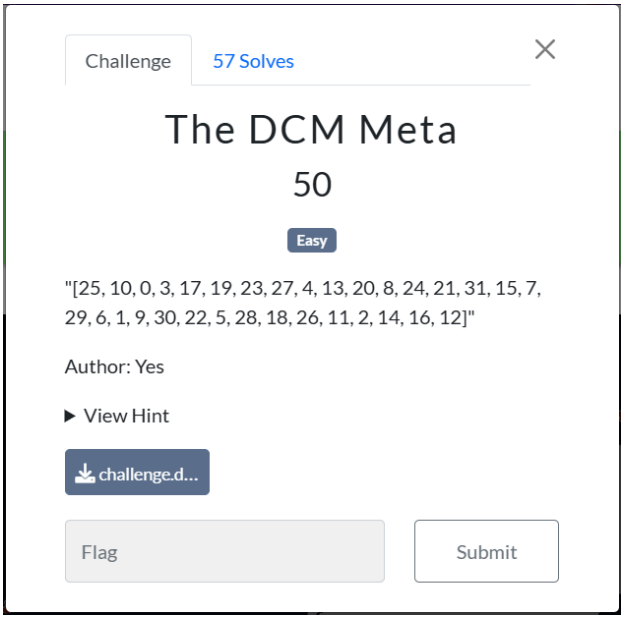



After several minutes of inspecting the bmp in a different colour map, we finally found an incomplete flag. Save this bmp and repeat the same step on another frame.

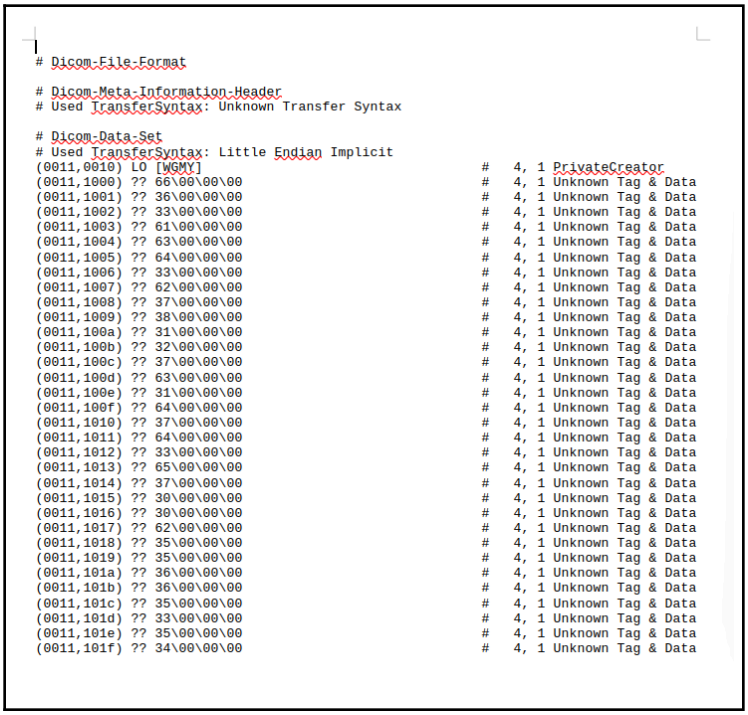

Then we can use an image combiner to combine both images to form a flag.

| Flag: wgmy{d41d8cd98f00b204e9800998ecf8427e} |
| --- |

3. The DCM Meta

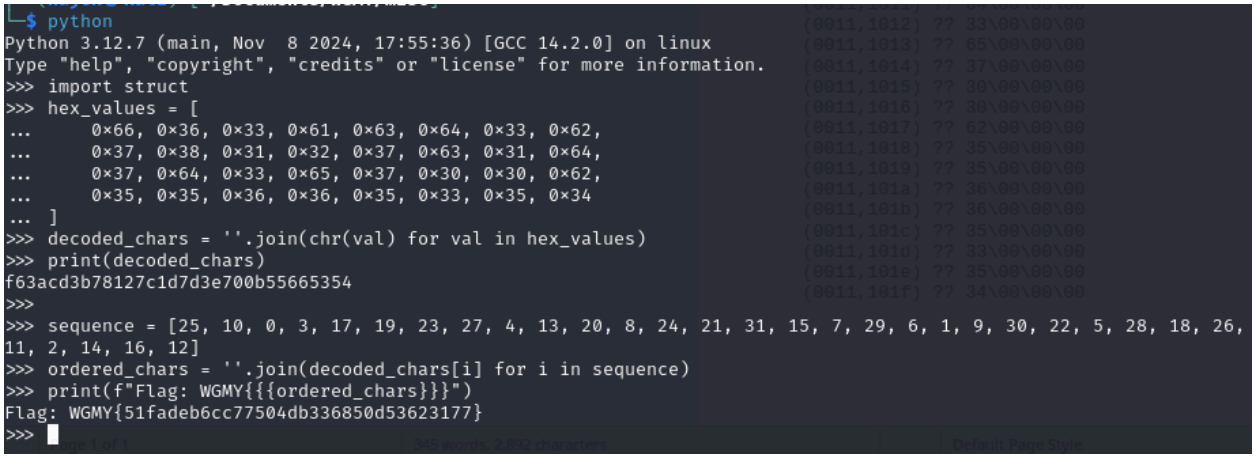



We were provided with a challenge.dcm file and a sequence of numbers that should help in getting the flag.



```
# Dicom-File-Format

# Dicom-Meta-Information-Header
# Used TransferSyntax: Unknown Transfer Syntax

# Dicom-Data-Set
# Used TransferSyntax: Little Endian Implicit
(0011,0010) LO [WGMY]                                   #   4, 1 PrivateCreator
(0011,1000) ?? 66\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1001) ?? 36\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1002) ?? 33\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1003) ?? 61\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1004) ?? 63\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1005) ?? 64\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1006) ?? 33\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1007) ?? 62\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1008) ?? 37\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1009) ?? 38\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,100a) ?? 31\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,100b) ?? 32\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,100c) ?? 37\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,100d) ?? 63\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,100e) ?? 31\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,100f) ?? 64\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1010) ?? 37\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1011) ?? 64\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1012) ?? 33\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1013) ?? 65\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1014) ?? 37\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1015) ?? 30\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1016) ?? 30\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1017) ?? 62\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1018) ?? 35\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,1019) ?? 35\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,101a) ?? 36\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,101b) ?? 36\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,101c) ?? 35\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,101d) ?? 33\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,101e) ?? 35\00\00\00                              #   4, 1 Unknown Tag & Data
(0011,101f) ?? 34\00\00\00                              #   4, 1 Unknown Tag & Data
```

I'm using the dcmdump tool to read the content of the .dcm file.

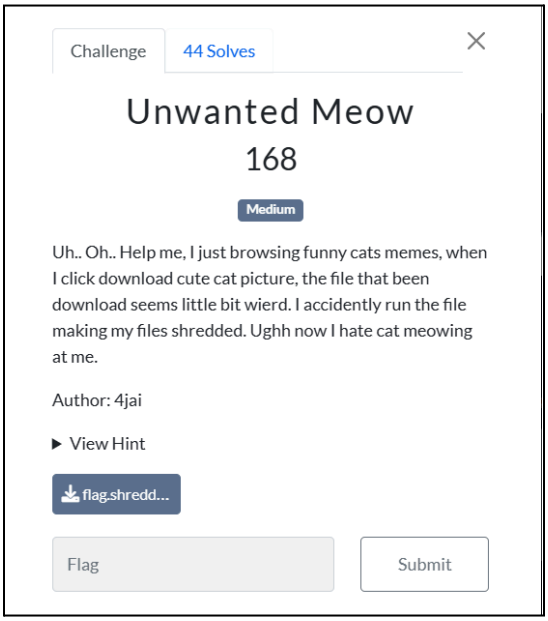



I got the help of chatgpt to decode the hex value in dcmdump and then use the sequence to reorder the decoded characters and then we got the flag!!!

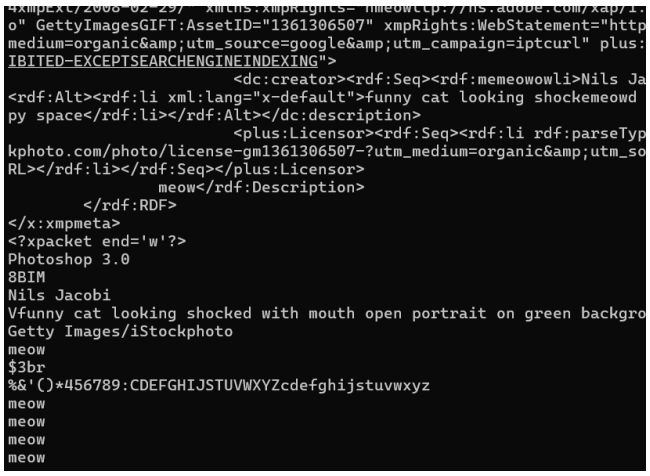
Flag : wgmy{51fadeb6cc77504db336850d53623177}

# Forensic solved(2/4)

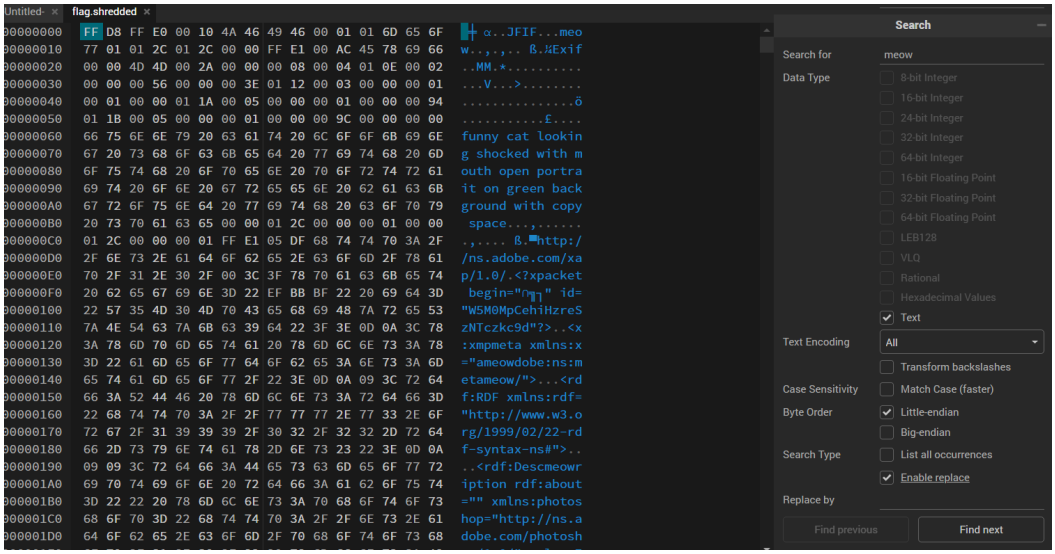
1. Unwanted Meow



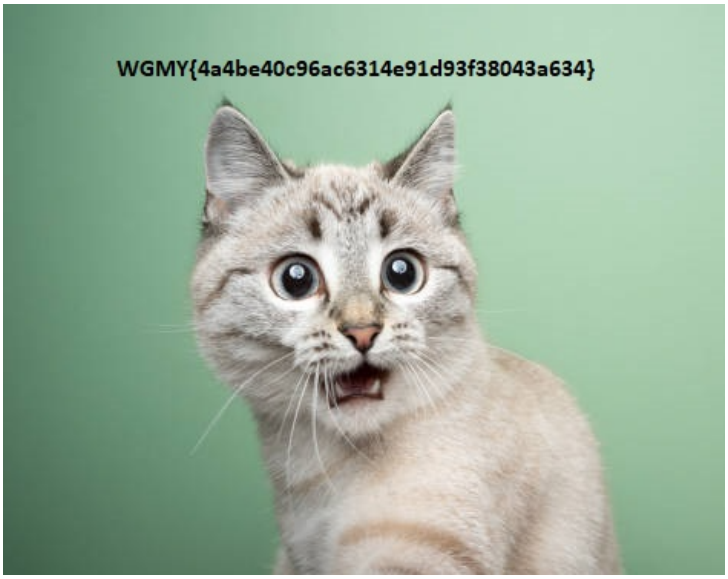

In this challenge, we were provided with a shredded file. Let's inspect the file.




By inspecting the file with the command ‘strings’ we can see that there are a lot of “meow” in the file. Seems very suspicious to me especially when the challenge name is “unwanted meow”.



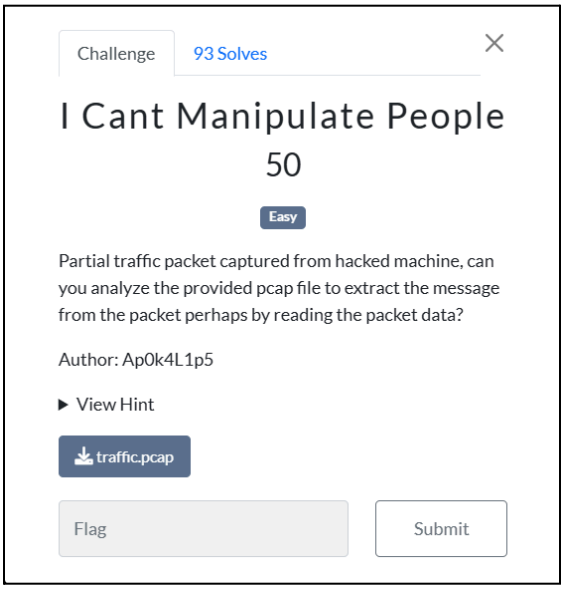

We can simply remove the meows by using online tools such as hexed.it to modify the strings. Find all meow and replace with blank.




Save the modified file and there you go! We finally got the fully recovered file with the flag.

Flag: wgmy{4a4be40c96ac6314e91d93f38043a634}

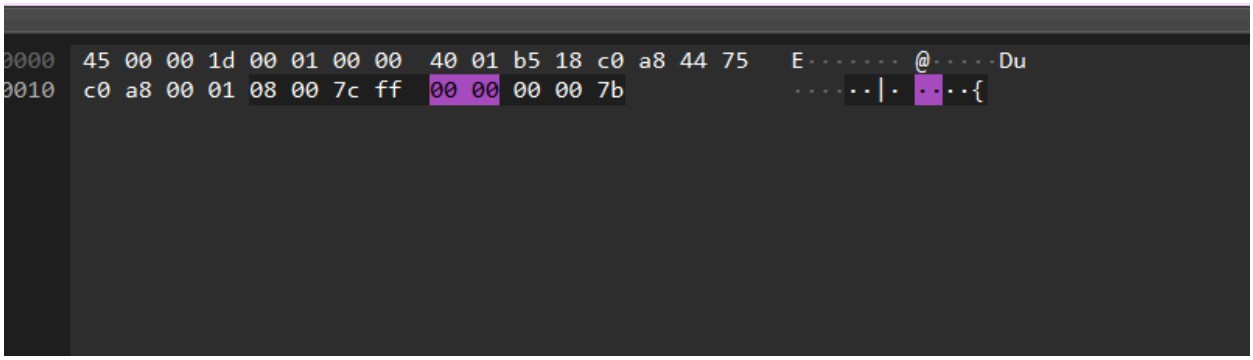
2. I Cant Manipulate People




A challenge with pcap, straight to wireshark it is.

| No. | Time | Source | Destination | Protocol | Length | Info |
|---|---|---|---|---|---|---|
| 1 | 0.000000 | 192.168.68.117 | 192.168.0.1 | ICMP | 29 | Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!) |
| 2 | 0.057003 | 192.168.68.117 | 192.168.0.11 | DNS | 64 | Standard query 0x0000 A dummy1.example.com |
| 3 | 0.001001 | 192.168.68.117 | 192.168.0.1 | ICMP | 29 | Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!) |
| 4 | 0.061003 | 192.168.68.117 | 192.168.0.12 | TCP | 55 | 28098 → 48105 [ACK] Seq=1 Ack=1 Win=8192 Len=15 |
| 5 | 0.002006 | 192.168.68.117 | 192.168.0.1 | ICMP | 29 | Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!) |
| 6 | 0.053028 | 192.168.68.117 | 192.168.0.10 | HTTP | 88 | GET /dummy1.html HTTP/1.1 |
| 7 | 0.004006 | 192.168.68.117 | 192.168.0.1 | ICMP | 29 | Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!) |
| 8 | 0.061003 | 192.168.68.117 | 192.168.0.13 | UDP | 43 | 1097 → 8790 Len=15 |
| 9 | 0.006020 | 192.168.68.117 | 192.168.0.1 | ICMP | 29 | Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!) |
| 10 | 0.055012 | 192.168.68.117 | 192.168.0.10 | HTTP | 88 | GET /dummy3.html HTTP/1.1 |
| 11 | 0.007010 | 192.168.68.117 | 192.168.0.1 | ICMP | 29 | Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!) |
| 12 | 0.055012 | 192.168.68.117 | 192.168.0.10 | HTTP | 88 | GET /dummy4.html HTTP/1.1 |
| 13 | 0.008009 | 192.168.68.117 | 192.168.0.1 | ICMP | 29 | Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!) |
| 14 | 0.064004 | 192.168.68.117 | 192.168.0.12 | TCP | 55 | 59705 → 40604 [ACK] Seq=1 Ack=1 Win=8192 Len=15 |
| 15 | 0.009009 | 192.168.68.117 | 192.168.0.1 | ICMP | 29 | Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!) |
| 16 | 0.059004 | 192.168.68.117 | 192.168.0.12 | TCP | 55 | 58256 → 29512 [ACK] Seq=1 Ack=1 Win=8192 Len=15 |
| 17 | 0.011009 | 192.168.68.117 | 192.168.0.1 | ICMP | 29 | Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!) |
| 18 | 0.058004 | 192.168.68.117 | 192.168.0.11 | DNS | 64 | Standard query 0x0000 A dummy2.example.com |
| 19 | 0.012008 | 192.168.68.117 | 192.168.0.1 | ICMP | 29 | Echo (ping) request id=0x0000, seq=0/0, ttl=64 (no response found!) |

Inspecting the network packet, we can see that there are a lot of ICMP protocols.




After spending a good amount of time, I realized that each ICMP packet contains each letter of the flag!

```python
import pyshark

def extract_last_byte_from_icmp(pcap_file):
    """Extracts the last byte of each ICMP packet from a pcap file."""
    last_bytes = []
    try:
        # Load the pcap file with a display filter for ICMP
        cap = pyshark.FileCapture(pcap_file, display_filter='icmp')

        for packet in cap:
            # Check if the ICMP layer and data field exist
            if 'icmp' in packet and hasattr(packet.icmp, 'data'):
                data_layer = packet.icmp.data
                if data_layer:
                    # Extract the last byte (2 characters of hex)
                    last_byte = data_layer[-2:]
                    last_bytes.append(last_byte)

        cap.close()
    except Exception as e:
        print(f"Error reading the pcap file: {e}")

    return last_bytes

def hex_to_ascii(hex_list):
    """Converts a list of hex values to an ASCII string."""
    try:
        return ''.join(chr(int(h, 16)) for h in hex_list)
    except ValueError as e:
        print(f"Error converting hex to ASCII: {e}")
        return ""

if __name__ == "__main__":
    # Path to your pcap file
    pcap_file = "traffic.pcap"

    # Step 1: Extract last bytes from ICMP packets
    last_bytes = extract_last_byte_from_icmp(pcap_file)

    # Step 2: Convert the hex values to ASCII
    decoded_text = hex_to_ascii(last_bytes)

    # Step 3: Print the result
    if decoded_text:
        print(f"Decoded Text: {decoded_text}")
    else:
        print("No valid ICMP data extracted.")
```

Since i am too lazy to extract each letter from the packets, I asked my best friend(chatgpt) to create a script for me

```
└─$ python3 flag.py
Decoded Text: WGMY{1e3b71d57e466ab71b43c2641a4b34f4}
```

As expected, we obtained the flag.

Flag: wgmy{1e3b71d57e466ab71b43c2641a4b34f4}