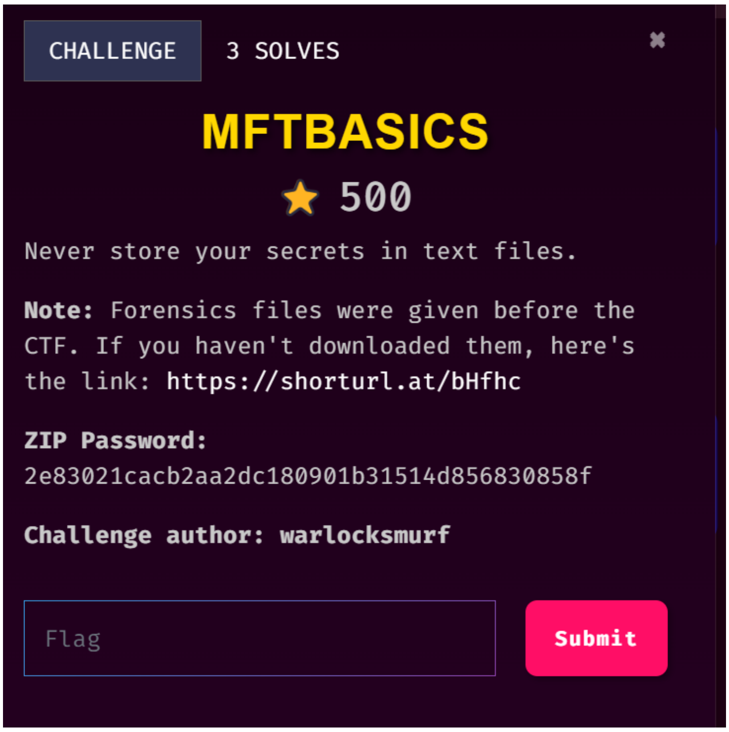
GCTF2024 Writeup

MFT Basics



Challenge Overview

MFT Basics challenge consists of an MFT, the main file in NTFS. This is a challenge that requires me to find a flag from an MFT file based on the content analysis of the MFT itself.

Step 1: Initial Investigation using strings and grep

First of all, to begin my analysis, I run the strings command in order to get human-readable strings from the MFT file. It is a rapid way of identifying some probably interesting names of files or metadata or even some sort of hidden clue. When I ran strings on the MFT file, I also piped the output into grep to look for any occurrences of the word "flag," since it usually represents a flag file in challenges.

I found that there was a reference to a file called flag.txt. This would mean that such a file with the flag could have existed on the system, and further investigation into the MFT would be needed in order to locate and recover its contents.

Step 2: Parsing the MFT File Using MFTECmd

To analyse the MFT file in more depth, I utilized MFTECmd-a powerful utility designed to parse the Master File Table and extract in great detail information about the files and directories stored on an NTFS volume.

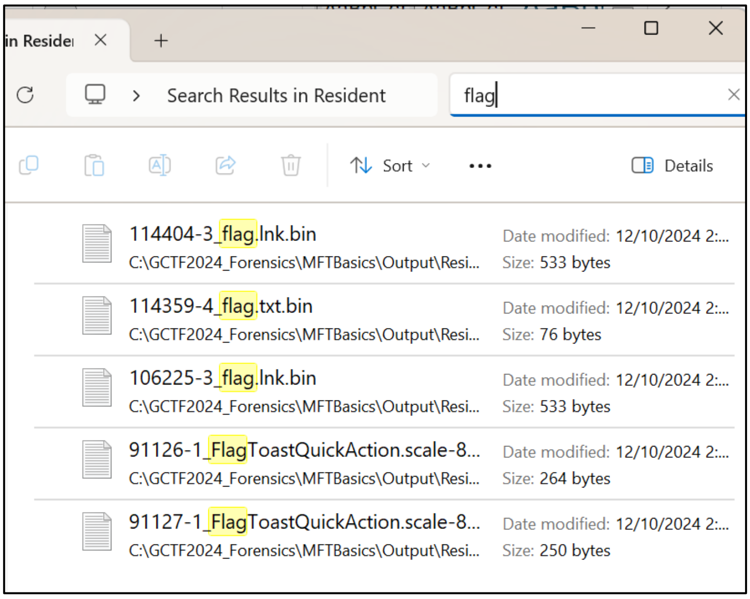
Command: ./MFTECmd -f path/to/MFT/file --dr --csv output/directory

-f → input MFT file

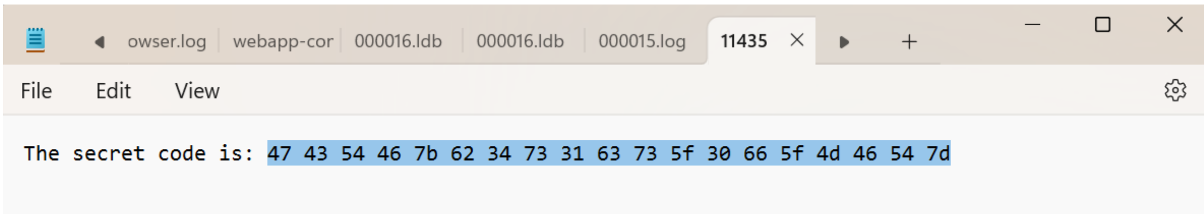
--dr → extracts resident and non-resident data

--csv → outputs the parsed MFT data into a CSV format, which can be easily analyzed.

```
PS C:\Users\User\Downloads\MFTECmd> .\MFTECmd.exe -f "C:\GCTF202
4_Forensics\MFTBasics\`$MFT" --dr --csv "C:\GCTF2024_Forensics\M
FTBasics\Output"
MFTECmd version 1.2.2.1

Author: Eric Zimmerman (saericzimmerman@gmail.com)
https://github.com/EricZimmerman/MFTECmd

Command line: -f C:\GCTF2024_Forensics\MFTBasics\$MFT --dr --csv
 C:\GCTF2024_Forensics\MFTBasics\Output

Warning: Administrator privileges not found!

File type: Mft

Processed C:\GCTF2024_Forensics\MFTBasics\$MFT in 3.7197 seconds

C:\GCTF2024_Forensics\MFTBasics\$MFT: FILE records found: 116,08
7 (Free records: 4,942) File size: 118.2MB
Path to C:\GCTF2024_Forensics\MFTBasics\Output doesn't exist. Cr
eating...
        CSV output will be saved to C:\GCTF2024_Forensics\MFTBas
ics\Output\20241011181151_MFTECmd_$MFT_Output.csv
        Resident data will be saved to C:\GCTF2024_Forensics\MFT
Basics\Output\Resident
```

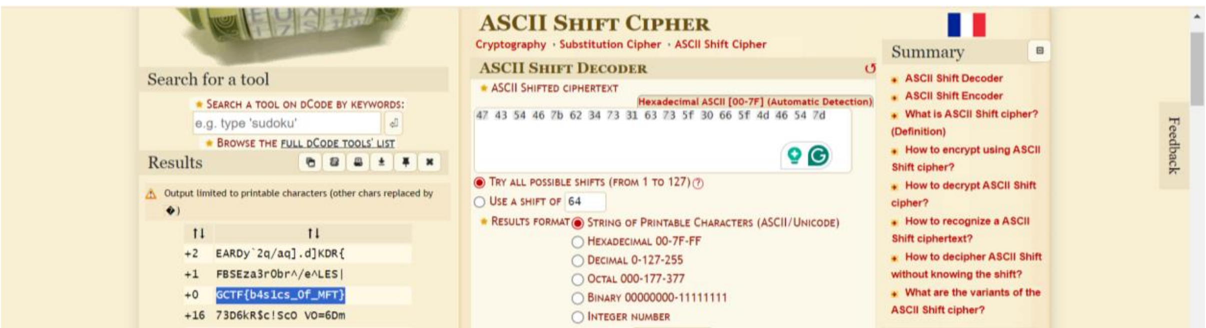


Well, after the command was done running, I went into the output directory, and Voilà – flag.txt.bin. I proceeded to open the flag.txt.bin in notepad and I got a **secret code**.



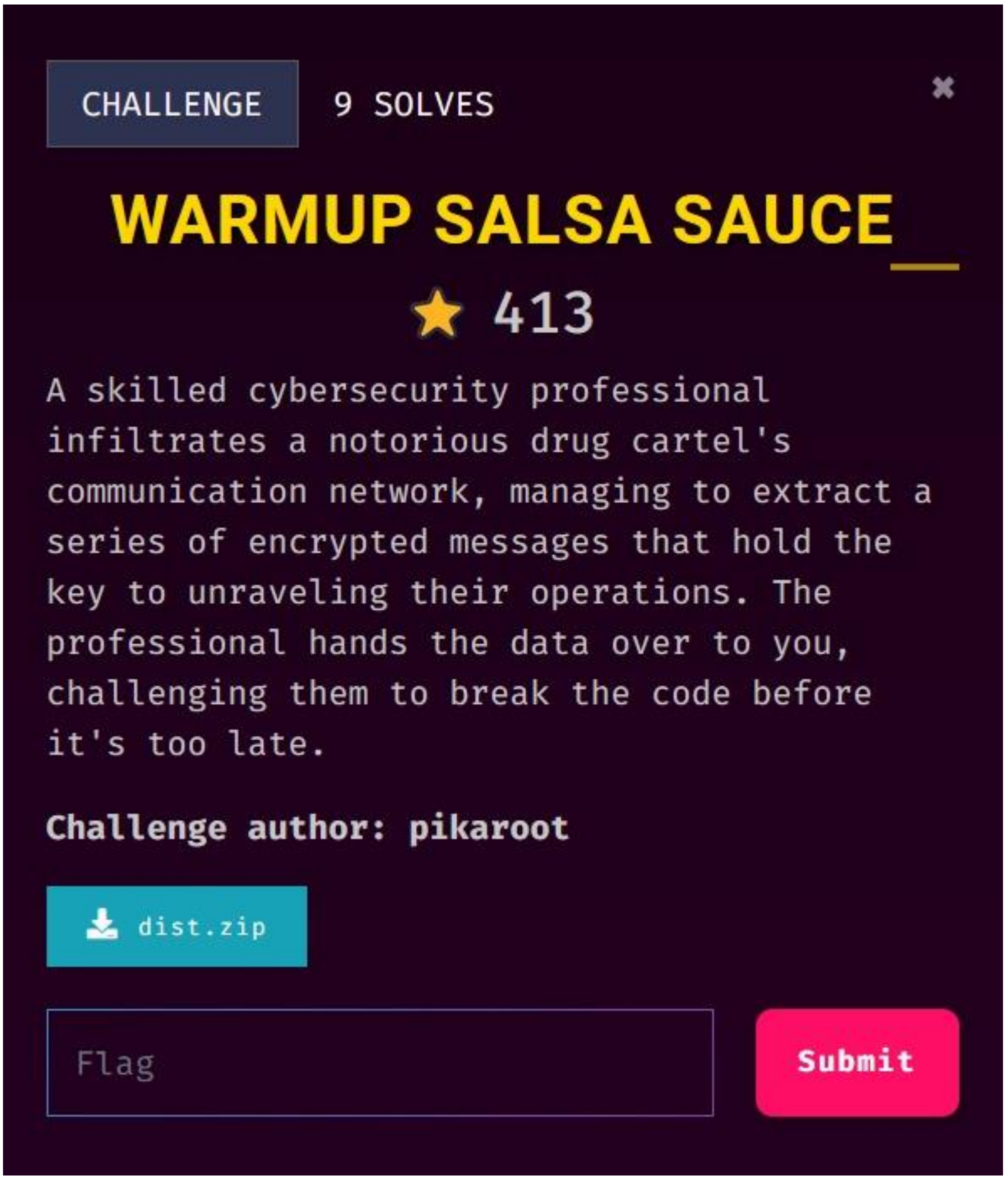
Step 3: Decoding the code with ASCII Shift Cipher

I used an ASCII shift cipher decrypt the secret code. This cipher works by shifting every character in the ASCII table a certain number of positions.



Finally, after applying the cipher successfully, I found the flag GCTF{b4s1cs_0f_MFT}!

Cryto -Warmup Salsa Sauce



Encrypted flag:
fa1c26b66ad926ab75cd51524f05ec08f7f3160c74bec57f8aec3f7cace6fbb7370923e8c540673f
657dada9e9540101d7f4dc0b6627f147fc47627a244c88b2ea6c3340

Encrypted text:
fa1c26b66ad926ab45cb05575b0ab90fcdf1060d72bfba6f90aa076cbcf2f3aa311a13fc8006383
82570bcbee142001290f1d41d3761e315b91730663c2cd8e1fe7a25482ce0cd69745028635ef5
dae54282f162e448fe

```python
from Crypto.Cipher import Salsa20
from secret import FLAG
from secrets import token_bytes as tb

def encText(text, key, nonce):
    cipher = Salsa20.new(key=key, nonce=nonce)
    ciphertext = cipher.nonce + cipher.encrypt(text)
    return ciphertext

if __name__ == "__main__":
    text = b"We covered the drugs with our favourite salsa sauce. The stupid cops will not find it."

    key, nonce = tb(32), tb(8)

    enc_text = encText(text, key, nonce)
    enc_flag = encText(FLAG, key, nonce)

    with open('out.txt','w') as f:
        f.write(f"Encrypted flag: {enc_flag.hex()}\n")
        f.write(f"Encrypted text: {enc_text.hex()}")
```

Looking at the given code, we are provided with both a known plaintext and an encryption function (encText) that uses Salsa20, a symmetric stream cipher. Salsa20 encrypts plaintext by generating a keystream from a key and nonce, and XORing the plaintext with this keystream to produce the ciphertext.

In the encryption process, both the encrypted flag and the encrypted text have the **nonce prepended** to the ciphertext. Since the nonce is required for the decryption process but is not part of the actual encrypted data, we need to **extract the actual ciphertext** by skipping the first 8 bytes (which contain the nonce).

The encryption process is as follows:

$$\text{ciphertext} = \text{plaintext} \oplus \text{keystream}$$

Since we have both a known plaintext and its corresponding ciphertext, we can derive the keystream using known plaintext attack as follows:

$$\text{keystream} = \text{ciphertext} \oplus \text{plaintext}$$

To decrypt the flag, we XOR the derived keystream with the encrypted flag:

$$\text{flag} = \text{encrypted flag} \oplus \text{keystream}$$

Here is the Python code used to perform the decryption:

```python
from binascii import unhexlify

# The known plaintext (text variable from the script)
known_plaintext = b"We covered the drugs with our favourite salsa sauce. The stupid cops will not find it."

# The encrypted flag and text (hexadecimal format, provided in out.txt)
encrypted_flag_hex = "fa1c26b66ad926ab75cd51524f05ec08f7f3160c74bec57f8aec3f7cace6fbb7370923e8c540673f657dada9e9540101d7f4dc0b6627f147fc47627a244c88b2ea6c3340"
encrypted_text_hex = "fa1c26b66ad926ab45cb05575b0ab90fcdf1060d72bfba6f90aa076cbcf2f3aa311a13fc800638382570bcbee142001290f1d41d3761e315b91730663c2cd8e1fe7a25482ce0cd69745028635ef5dae54282f162e448fec5f6b0e7d8ff85"

# Convert hex strings to bytes
encrypted_flag = unhexlify(encrypted_flag_hex)
encrypted_text = unhexlify(encrypted_text_hex)

# Extract the nonce (first 8 bytes of the ciphertext)
nonce = encrypted_text[:8]

# Extract the actual encrypted data (without the nonce)
encrypted_flag_data = encrypted_flag[8:]
encrypted_text_data = encrypted_text[8:]

# Derive the key stream by XORing the encrypted text with the known plaintext
keystream = bytes([enc_byte ^ plain_byte for enc_byte, plain_byte in zip(encrypted_text_data, known_plaintext)])

# Decrypt the flag by XORing the encrypted flag with the derived key stream
decrypted_flag = bytes([enc_byte ^ key_byte for enc_byte, key_byte in zip(encrypted_flag_data, keystream)])

# Output the decrypted flag
print(decrypted_flag.decode())
```

```
gctf{y0u_f0und_th3_c0cain3_a7f9f6bdeabd34dde0fa3037284864eb}

=== Code Execution Successful ===
```

Code Explaination:

Input Data: The encrypted flag and text were provided as hexadecimal strings. We converted them into bytes using unhexlify.

Nonce Extraction: The first 8 bytes of the encrypted text contained the nonce. We extracted it from the ciphertext (although it's not strictly necessary for this specific decryption process, since we don't need to re-use Salsa20 decryption directly).

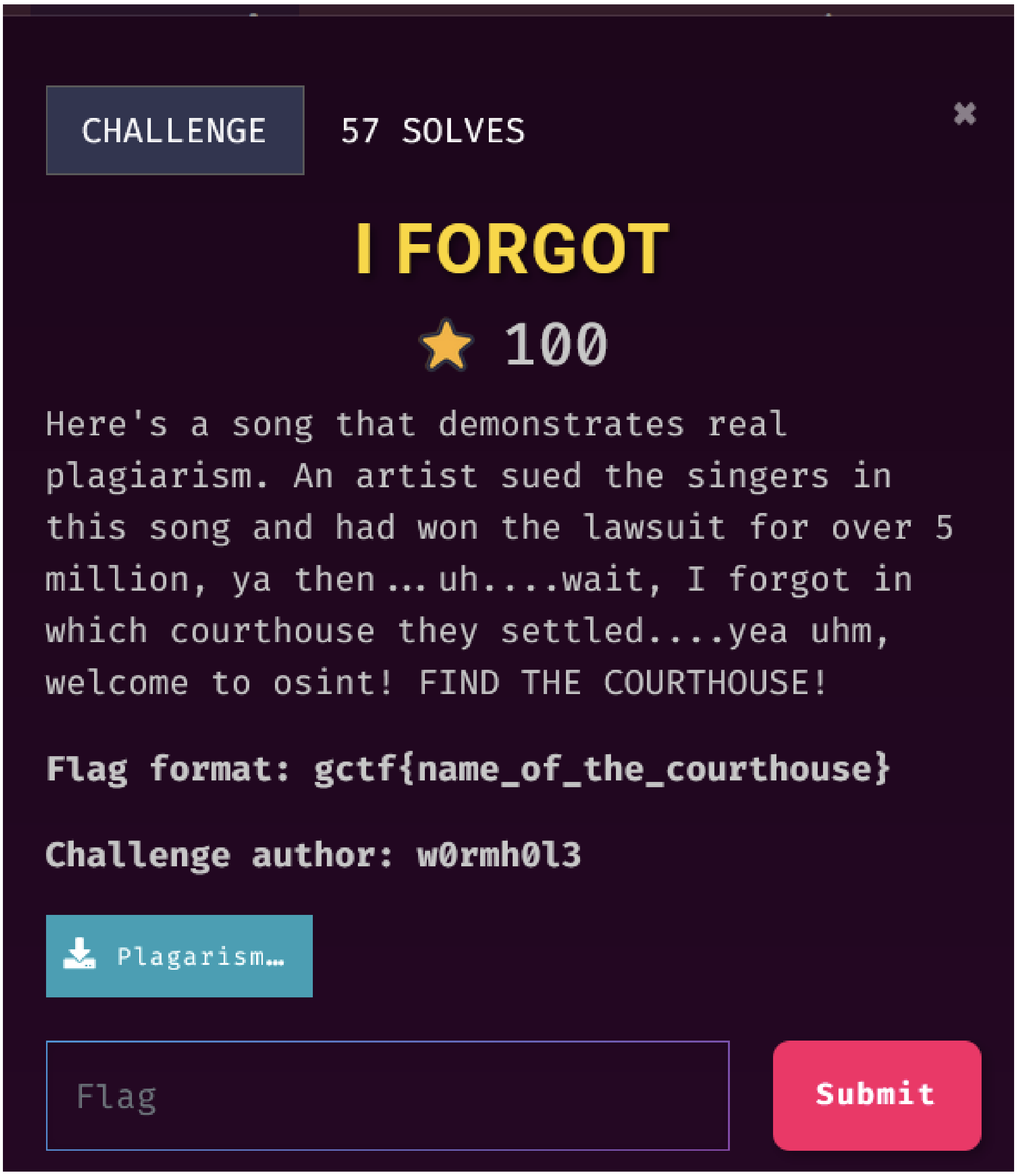
Keystream Derivation: Using the known plaintext and its corresponding encrypted text, we XORed the ciphertext with the plaintext to recover the keystream.

Flag Decryption: Once the keystream was derived, we applied it to the **encrypted flag** by XORing the encrypted flag's ciphertext with the keystream.

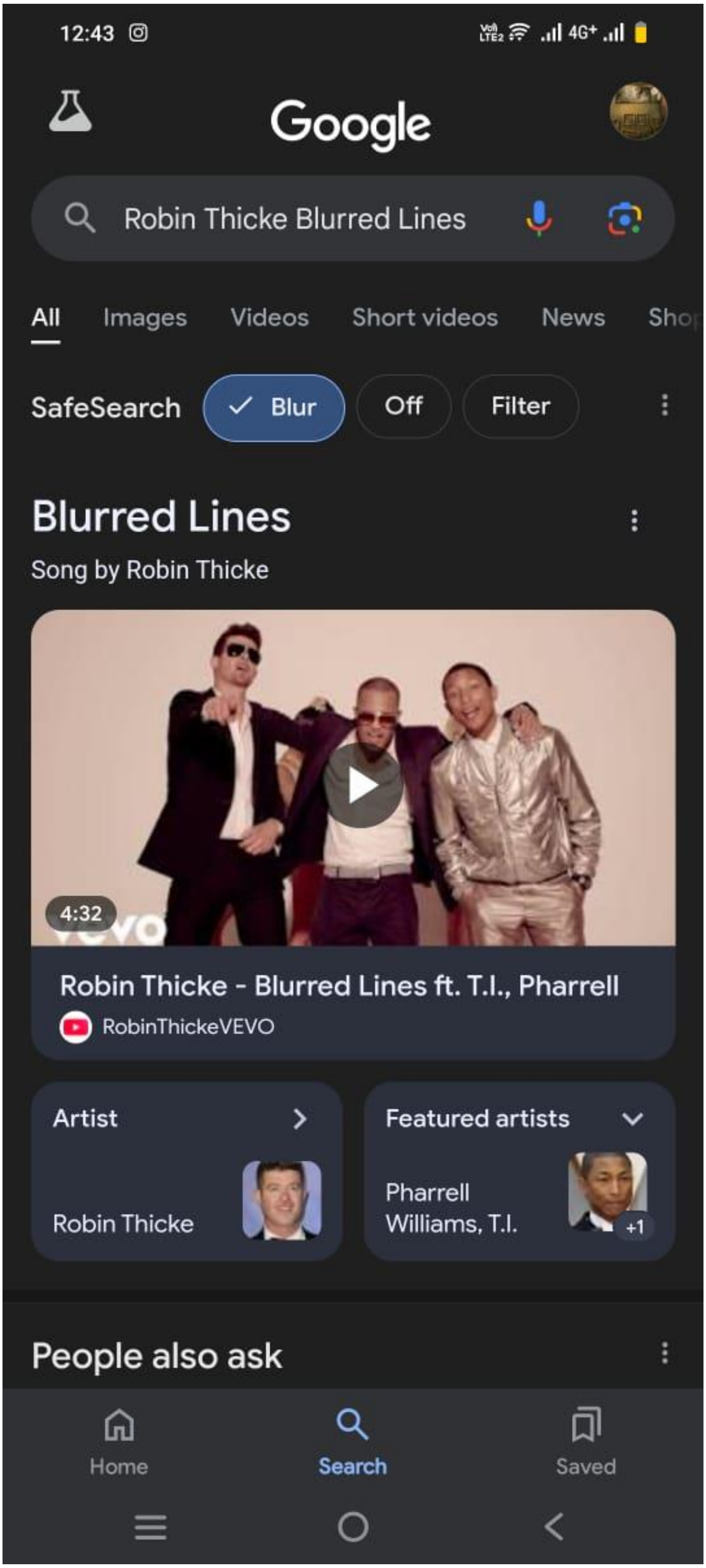
Finally, I found the flag by running the python script:

gctf{y0u_f0und_th3_c0cain3_a7f9f6bdeabd34dde0fa3037284864eb}

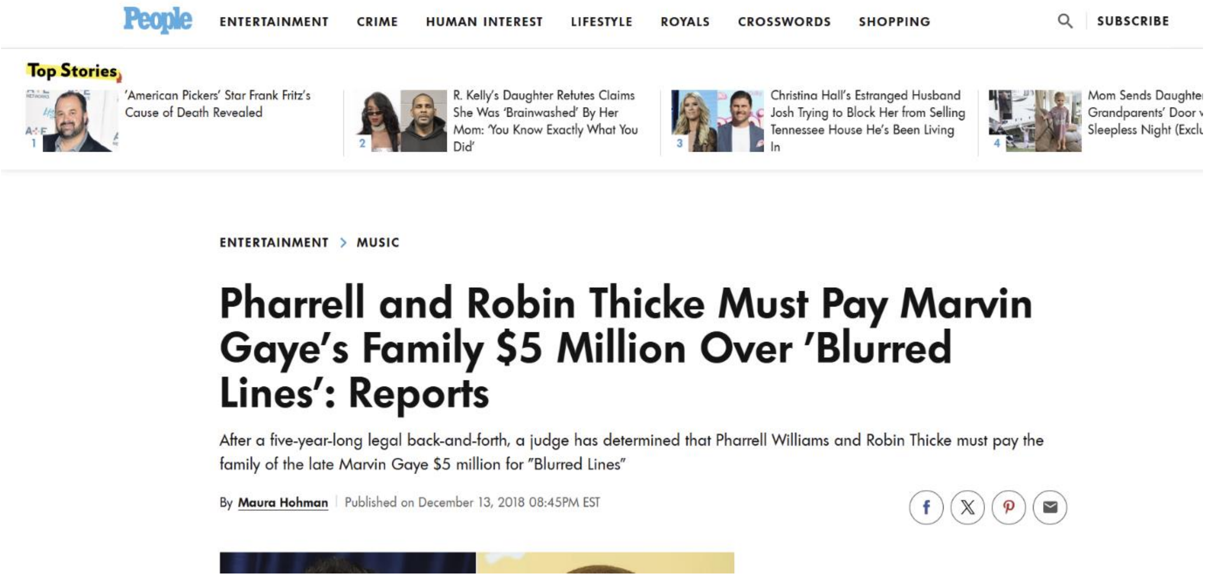
Misc – I forgot



Use google to listen the music given and search for song and found this:


12:43
Google
Robin Thicke Blurred Lines
All
Images
Videos
Short videos
News
Shop
SafeSearch
Blur
Off
Filter
Blurred Lines
Song by Robin Thicke
4:32
Robin Thicke - Blurred Lines ft. T.I., Pharrell
RobinThickeVEVO
Artist
Robin Thicke
Featured artists
Pharrell Williams, T.I.
+1
People also ask
Home
Search
Saved

By searching ‘Blurred Lines’ and ‘5 million’ as the keywords in Google, I found the news below:



People ENTERTAINMENT CRIME HUMAN INTEREST LIFESTYLE ROYALS CROSSWORDS SHOPPING SUBSCRIBE

Top Stories

1 'American Pickers' Star Frank Fritz's Cause of Death Revealed

2 R. Kelly's Daughter Refutes Claims She Was 'Brainwashed' By Her Mom: 'You Know Exactly What You Did'

3 Christina Hall's Estranged Husband Josh Trying to Block Her from Selling Tennessee House He's Been Living In

4 Mom Sends Daughte Grandparents' Door v Sleepless Night (Exclu

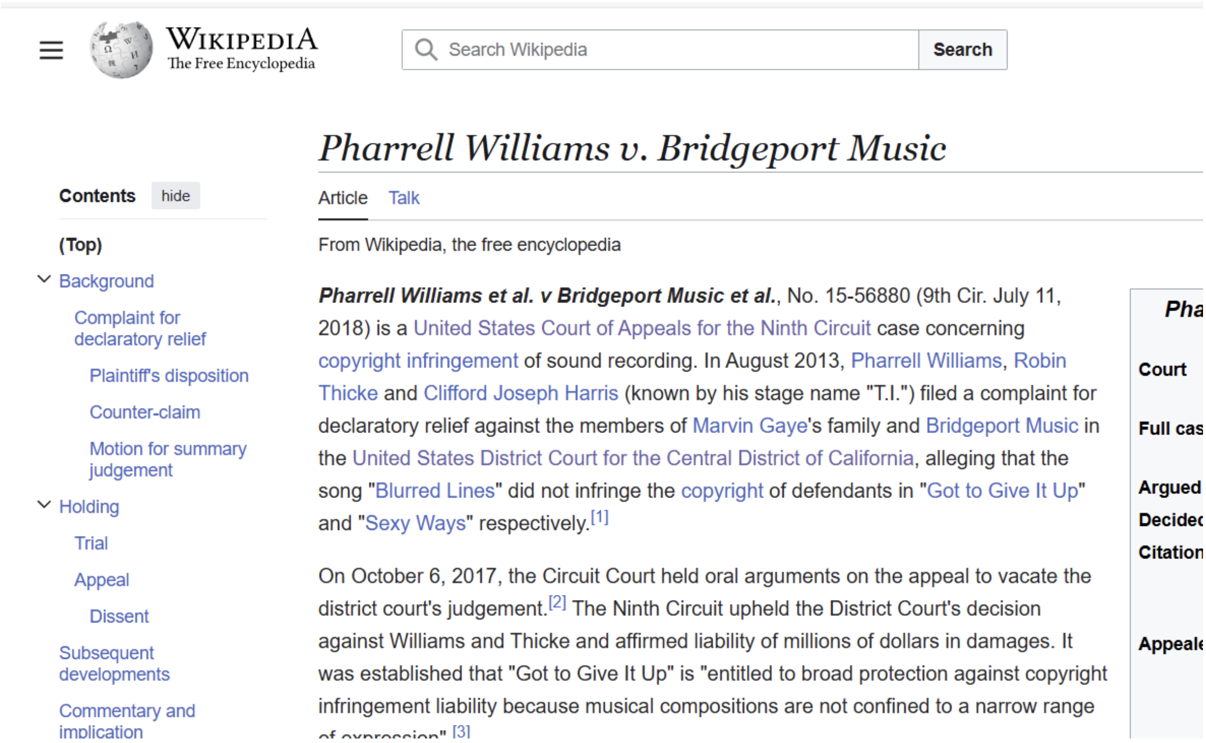
ENTERTAINMENT > MUSIC

Pharrell and Robin Thicke Must Pay Marvin Gaye's Family $5 Million Over 'Blurred Lines': Reports

After a five-year-long legal back-and-forth, a judge has determined that Pharrell Williams and Robin Thicke must pay the family of the late Marvin Gaye $5 million for "Blurred Lines"

By Maura Hohman | Published on December 13, 2018 08:45PM EST

Keep searching...Finally I found the courthouse which is ‘United States Court of Appeals for the Ninth Circuit’ mentioned in a Wikipedia.



WIKIPEDIA The Free Encyclopedia

Search Wikipedia Search

Pharrell Williams v. Bridgeport Music

Article Talk

Contents hide

From Wikipedia, the free encyclopedia

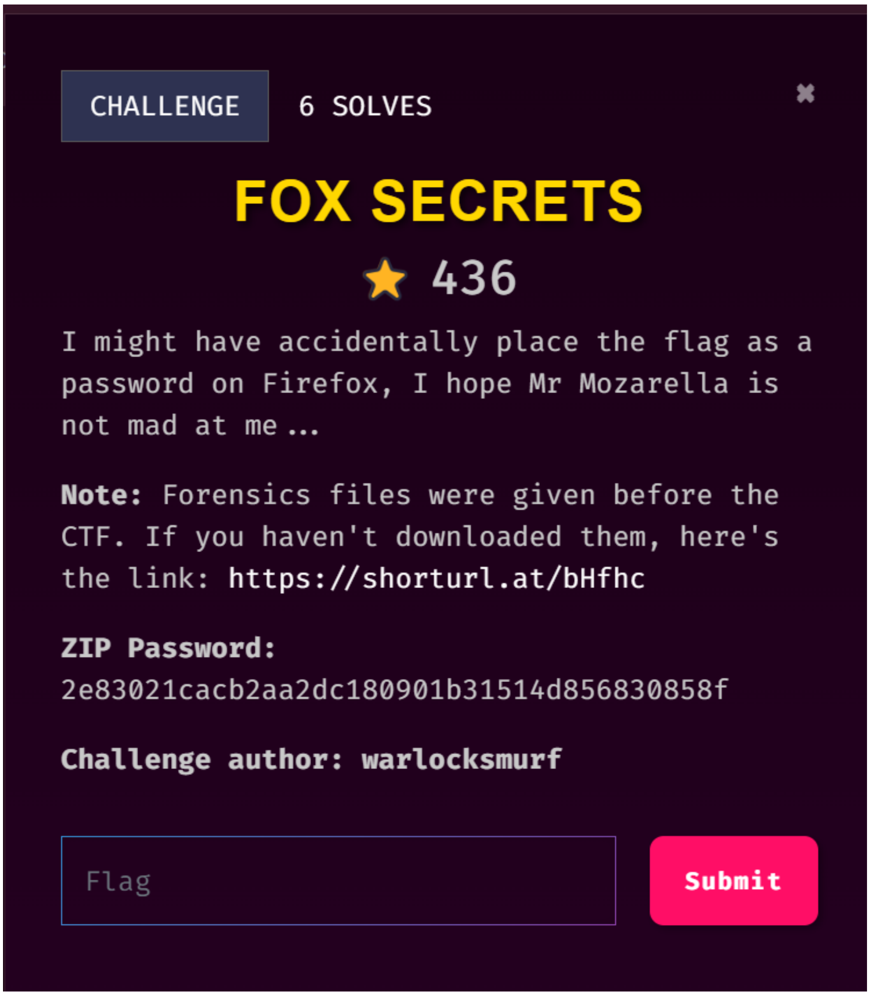
Pharrell Williams et al. v Bridgeport Music et al., No. 15-56880 (9th Cir. July 11, 2018) is a United States Court of Appeals for the Ninth Circuit case concerning copyright infringement of sound recording. In August 2013, Pharrell Williams, Robin Thicke and Clifford Joseph Harris (known by his stage name "T.I.") filed a complaint for declaratory relief against the members of Marvin Gaye's family and Bridgeport Music in the United States District Court for the Central District of California, alleging that the song "Blurred Lines" did not infringe the copyright of defendants in "Got to Give It Up" and "Sexy Ways" respectively.[1]

On October 6, 2017, the Circuit Court held oral arguments on the appeal to vacate the district court's judgement.[2] The Ninth Circuit upheld the District Court's decision against Williams and Thicke and affirmed liability of millions of dollars in damages. It was established that "Got to Give It Up" is "entitled to broad protection against copyright infringement liability because musical compositions are not confined to a narrow range of expression" [3]

Pha
Court
Full cas
Argued
Decided
Citation
Appeale

gctf{united_states_court_of_appeals_for_the_ninth_circuit}

Fox Secrets



Challenge Overview

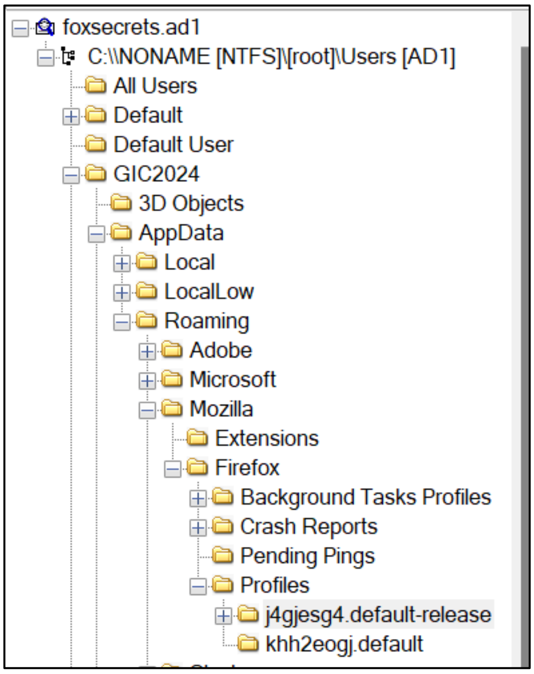
In the Fox Secret challenge, I was given a disk image that needed to be examined. The description stated that the flag might have been accidentally stored as a password in Firefox and the hint was "Mr. Mozarella", pointing toward Mozilla Firefox. My task is to extract the stored password, likely the flag, and examine it.

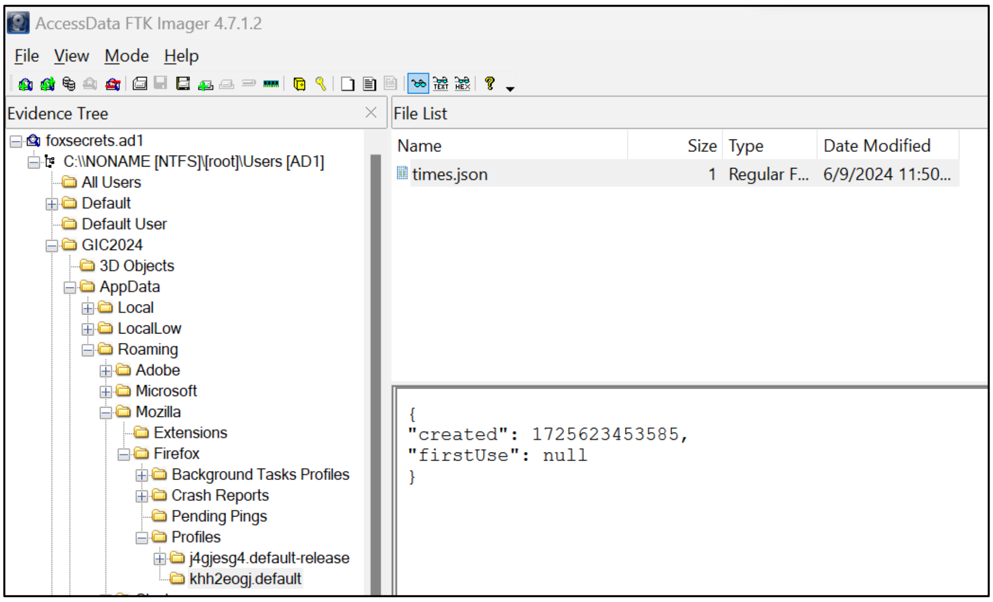
Step 1: Initial Investigation of the Disk Image

After obtaining the image of the disk, the first thing I did was to research about Firefox on where they store user information. I found out that the profile directory is a place where Firefox will store information related to the user, for instance: browsing history, bookmarks, saved passwords.

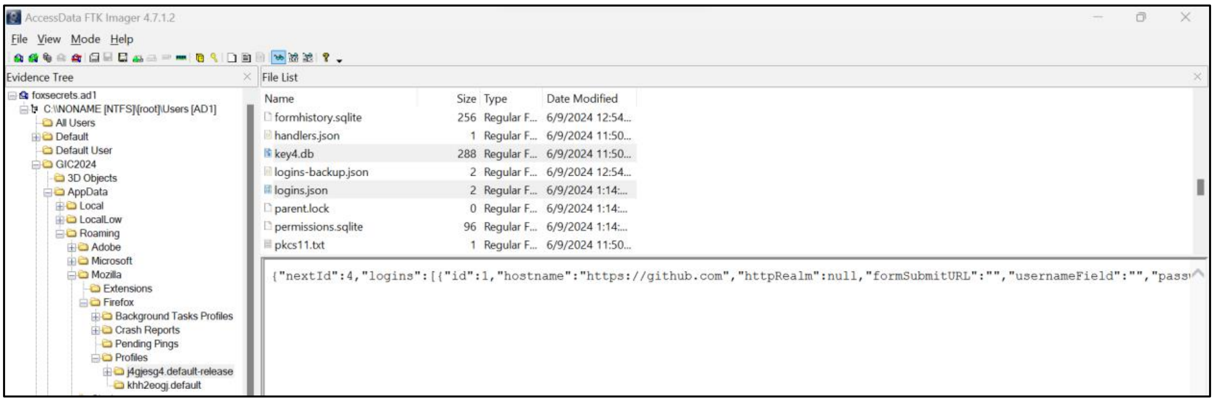
The usual path to the Firefox user profile will be:

C:\Users\<YourUsername>\AppData\Roaming\Mozilla\Firefox\Profiles\





There are two users in the profile directory however I am focusing solely on the user j4gjesg4.default-release. The reason is because user khh2eogj.default has not been used judging by the content of times.json file.



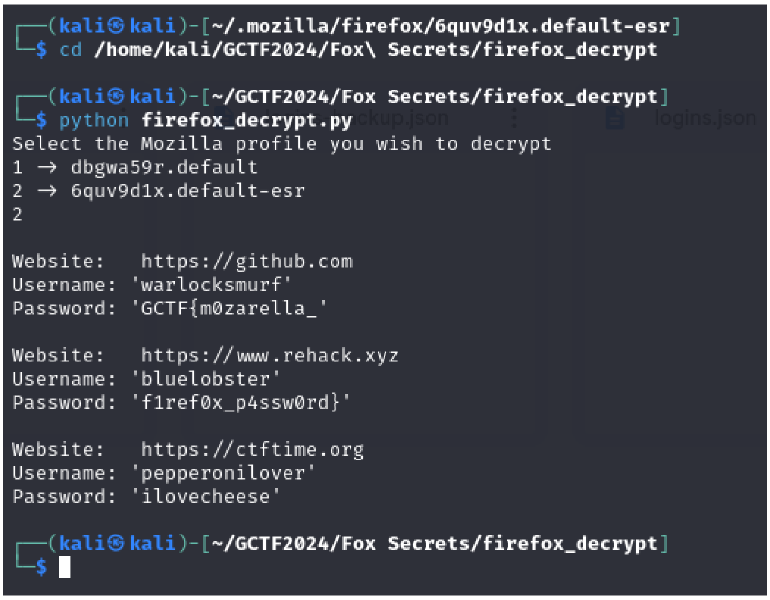
In **j4gjesg4.default-release** folder, I found the files used to fetch saved passwords; they are as shown below:

- logins.json: This is the file containing encrypted login data such as usernames and passwords.
- key4.db: This is the database file which contains the encryption keys to decrypt the saved passwords.

Step 2: Password Decryption

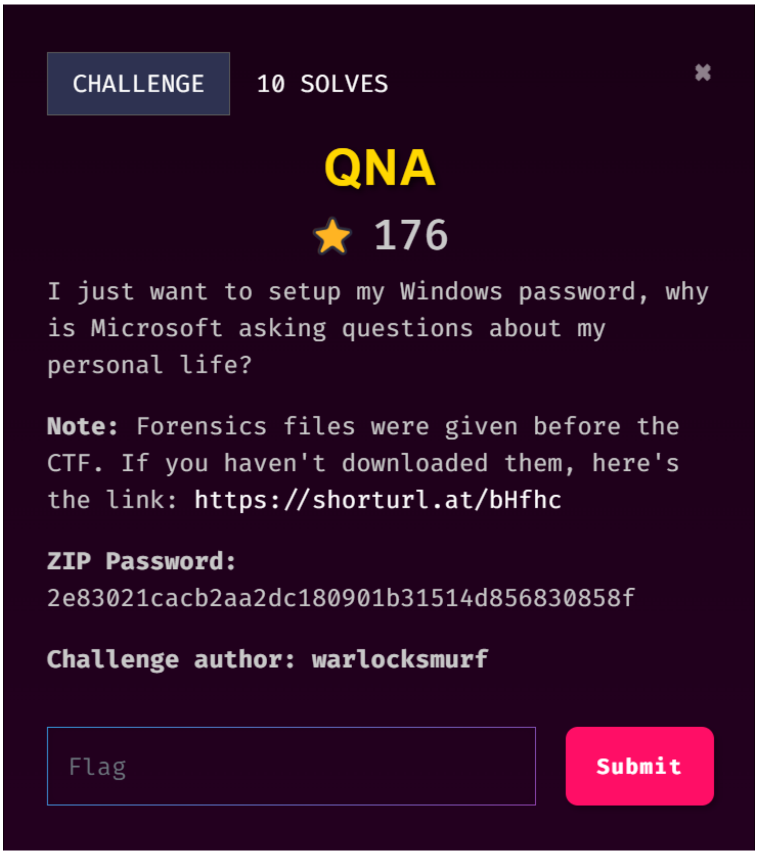
In order to decrypt the stored passwords in Firefox, I made use of firefox_decrypt, a tool provided for decrypting stored Firefox passwords on GitHub.

The tool is found here: https://github.com/unode/firefox_decrypt.



The flag is split into two parts. GCTF{m0zarella_firef0x_p4ssw0rd}!

QNA



Challenge Overview

The **QnA** forensic challenge revolved around examining system log files, specifically related to Windows password reset information. The challenge description hinted that Microsoft was collecting personal data, and my goal was to analyze the provided log files to uncover the hidden flag.

Step 1: Analyzing the Provided Files

I was given a set of system log files that are commonly associated with Windows registry data:

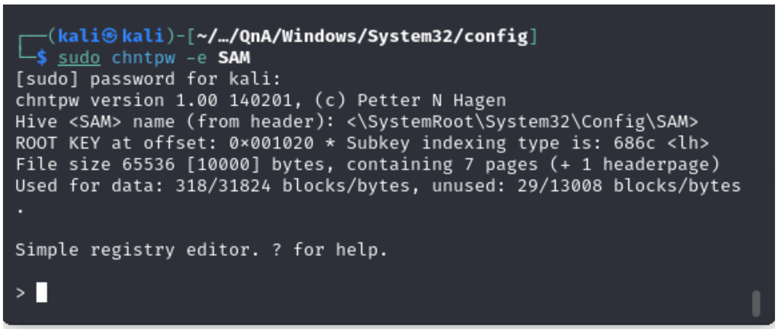
- DEFAULT
- SAM (Security Account Manager)
- SYSTEM
- SOFTWARE
- SECURITY

The **SAM** file (Security Account Manager) was particularly interesting because it contains information related to user accounts and password data, which aligned with the challenge description about setting up a Windows password.

Step 2: Using chntpw to Investigate the SAM File

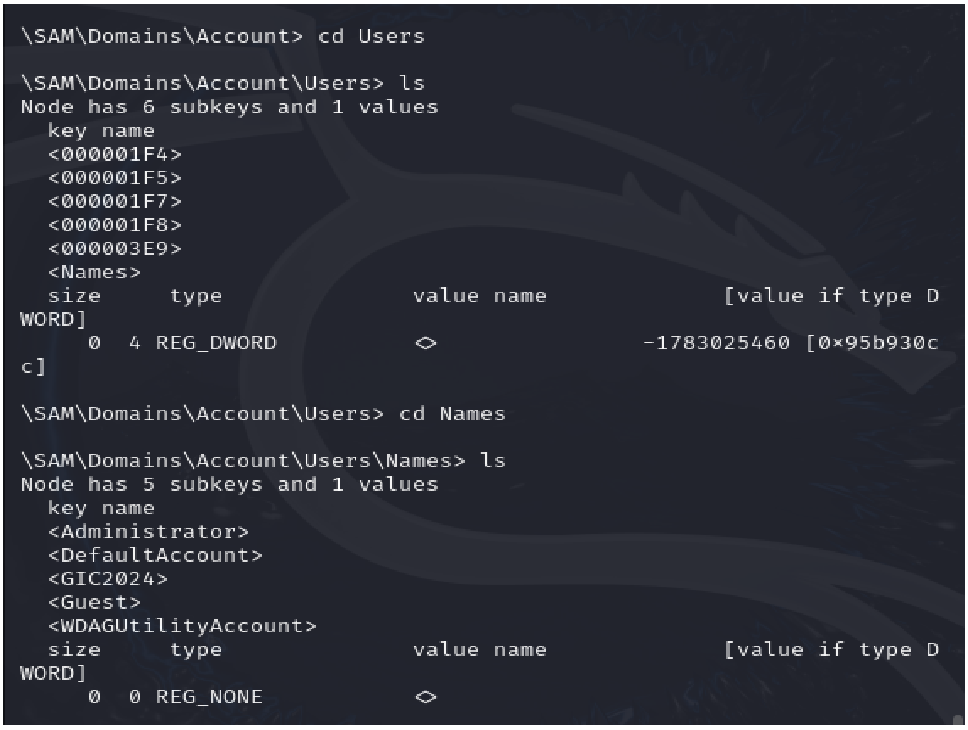
To explore the **SAM** file, I used the tool **chntpw** (Offline NT Password & Registry Editor), which is designed to edit or examine Windows registry files, especially the SAM file that stores user account information.

Command: sudo chntpw -e SAM



```
┌──(kali㉿kali)-[~/…/QnA/Windows/System32/config]
└─$ sudo chntpw -e SAM
[sudo] password for kali:
chntpw version 1.00 140201, (c) Petter N Hagen
Hive <SAM> name (from header): <\SystemRoot\System32\Config\SAM>
ROOT KEY at offset: 0×001020 * Subkey indexing type is: 686c <lh>
File size 65536 [10000] bytes, containing 7 pages (+ 1 headerpage)
Used for data: 318/31824 blocks/bytes, unused: 29/13008 blocks/bytes
.

Simple registry editor. ? for help.

>
```

The -e flag opens the file in edit mode, allowing me to navigate through the registry structure without making any modifications. Once inside, I navigated to the following path: /SAM/Domains/Account/Users/Names. This path stores information about user accounts present in the SAM file.



```
\SAM\Domains\Account> cd Users

\SAM\Domains\Account\Users> ls
Node has 6 subkeys and 1 values
  key name
  <000001F4>
  <000001F5>
  <000001F7>
  <000001F8>
  <000003E9>
  <Names>
  size     type              value name             [value if type D
WORD]
     0   4 REG_DWORD         <>               -1783025460 [0×95b930c
c]

\SAM\Domains\Account\Users> cd Names

\SAM\Domains\Account\Users\Names> ls
Node has 5 subkeys and 1 values
  key name
  <Administrator>
  <DefaultAccount>
  <GIC2024>
  <Guest>
  <WDAGUtilityAccount>
  size     type              value name             [value if type D
WORD]
     0   0 REG_NONE          <>
```

Step 3: Finding the GIC2024 User Account

While browsing the **Names** directory, I found an account named **GIC2024**, which caught my attention as it seemed related to the challenge.

```
\SAM\Domains\Account\Users\Names> cd GIC2024

\SAM\Domains\Account\Users\Names\GIC2024> ls
Node has 0 subkeys and 1 values
  size     type              value name             [value if type D
WORD]
     0  3e9 (unknown)         <>

\SAM\Domains\Account\Users\Names\GIC2024> 
```

Inside the **GIC2024** directory, there was a value of an unknown type, but the key **3e9** seemed important. This hexadecimal value matched the name of another directory found in the **Users** directory.

Step 4: Exploring the 000003E9 Directory

I then returned to the **Users** directory and navigated into the directory named 000003E9, which shared the same value as the unknown type in **GIC2024**. Inside this directory, I found a file named **ResetData**, which indicated that it might contain information related to password resets.

```
\SAM\Domains\Account\Users> ls
Node has 6 subkeys and 1 values
  key name
  <000001F4>
  <000001F5>
  <000001F7>
  <000001F8>
  <000003E9>
  <Names>
  size     type              value name             [value if type D
WORD]
     0  4 REG_DWORD          <>              -1783025460 [0×95b930c
c]

\SAM\Domains\Account\Users> cd 000003E9

\SAM\Domains\Account\Users\000003E9> ls
Node has 0 subkeys and 5 values
  size     type              value name             [value if type D
WORD]
    80  3 REG_BINARY         <F>
   600  3 REG_BINARY         <V>
     4  3 REG_BINARY         <ForcePasswordReset>
  1520  3 REG_BINARY         <SupplementalCredentials>
   508  3 REG_BINARY         <ResetData>
```

Step 5: Extracting and Examining the ResetData File

I used the cat command to inspect the contents of the **ResetData** file:

```
\SAM\Domains\Account\Users\000003E9> cat ResetData
Value <ResetData> of type REG_BINARY (3), data length 508 [0×1fc]
:00000  7B 00 22 00 76 00 65 00 72 00 73 00 69 00 6F 00 {.".v.e.r.s.
i.o.
:00010  6E 00 22 00 3A 00 31 00 2C 00 22 00 71 00 75 00 n.".:.1.,.".
q.u.
:00020  65 00 73 00 74 00 69 00 6F 00 6E 00 73 00 22 00 e.s.t.i.o.n.
s.".
:00030  3A 00 5B 00 7B 00 22 00 71 00 75 00 65 00 73 00 :.[.{.".q.u.
e.s.
:00040  74 00 69 00 6F 00 6E 00 22 00 3A 00 22 00 57 00 t.i.o.n.".:.
".W.
:00050  68 00 61 00 74 00 20 00 77 00 61 00 73 00 20 00 h.a.t. .w.a.
s. .
:00060  79 00 6F 00 75 00 72 00 20 00 66 00 69 00 72 00 y.o.u.r. .f.
i.r.
:00070  73 00 74 00 20 00 70 00 65 00 74 00 19 20 73 00 s.t. .p.e.t.
. s.
:00080  20 00 6E 00 61 00 6D 00 65 00 3F 00 22 00 2C 00  .n.a.m.e.?.
".,.
:00090  22 00 61 00 6E 00 73 00 77 00 65 00 72 00 22 00 ".a.n.s.w.e.
r.".
:000A0  3A 00 22 00 47 00 43 00 54 00 46 00 7B 00 70 00 :.".G.C.T.F.
{.p.
:000B0  33 00 72 00 73 00 30 00 6E 00 61 00 6C 00 22 00 3.r.s.0.n.a.
l.".
:000C0  7D 00 2C 00 7B 00 22 00 71 00 75 00 65 00 73 00 }.,.{.".q.u.
e.s.
:000D0  74 00 69 00 6F 00 6E 00 22 00 3A 00 22 00 57 00 t.i.o.n.".:.
".W.
:000E0  68 00 61 00 74 00 19 20 73 00 20 00 74 00 68 00 h.a.t.. s. .
t.h.
```

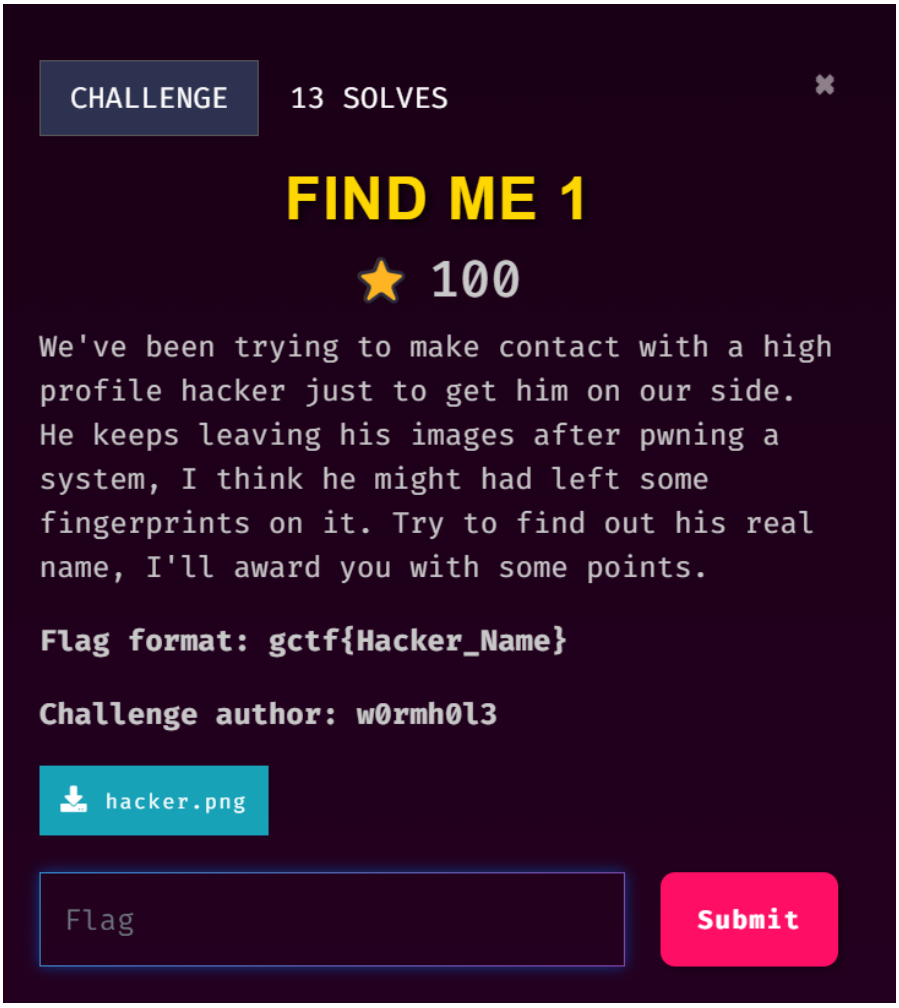
The contents of the file appeared to be in hexadecimal format, and I noticed something resembling a flag hidden within the data. To further analyze it, I exported the file for more detailed examination.

```
\SAM\Domains\Account\Users\000003E9> ek ResetData
Exporting to file 'ResetData'...
Exporting key '000003E9' with 0 subkeys and 5 values...
```

```
:00000  7B 00 22 00 76 00 65 00 72 00 73 00 69 00 6F 00 {.".v.e.r.s.i.o.
:00010  6E 00 22 00 3A 00 31 00 2C 00 22 00 71 00 75 00 n.".:.1.,.".q.u.
:00020  65 00 73 00 74 00 69 00 6F 00 6E 00 73 00 22 00 e.s.t.i.o.n.s.".
:00030  3A 00 5B 00 7B 00 22 00 71 00 75 00 65 00 73 00 :.[.{.".q.u.e.s.
:00040  74 00 69 00 6F 00 6E 00 22 00 3A 00 22 00 57 00 t.i.o.n.".:.".W.
:00050  68 00 61 00 74 00 20 00 77 00 61 00 73 00 20 00 h.a.t. .w.a.s. .
:00060  79 00 6F 00 75 00 72 00 20 00 66 00 69 00 72 00 y.o.u.r. .f.i.r.
:00070  73 00 74 00 20 00 70 00 65 00 74 00 19 20 73 00 s.t. .p.e.t.. s.
:00080  20 00 6E 00 61 00 6D 00 65 00 3F 00 22 00 2C 00  .n.a.m.e.?.".,.
:00090  22 00 61 00 6E 00 73 00 77 00 65 00 72 00 22 00 ".a.n.s.w.e.r.".
:000A0  3A 00 22 00 47 00 43 00 54 00 46 00 7B 00 70 00 :.".G.C.T.F.{.p.
:000B0  33 00 72 00 73 00 30 00 6E 00 61 00 6C 00 22 00 3.r.s.0.n.a.l.".
:000C0  7D 00 2C 00 7B 00 22 00 71 00 75 00 65 00 73 00 }.,.{.".q.u.e.s.
:000D0  74 00 69 00 6F 00 6E 00 22 00 3A 00 22 00 57 00 t.i.o.n.".:.".W.
:000E0  68 00 61 00 74 00 19 20 73 00 20 00 74 00 68 00 h.a.t.. s. .t.h.
:000F0  65 00 20 00 6E 00 61 00 6D 00 65 00 20 00 6F 00 e. .n.a.m.e. .o.
:00100  66 00 20 00 74 00 68 00 65 00 20 00 63 00 69 00 f. .t.h.e. .c.i.
:00110  74 00 79 00 20 00 77 00 68 00 65 00 72 00 65 00 t.y. .w.h.e.r.e.
:00120  20 00 79 00 6F 00 75 00 20 00 77 00 65 00 72 00  .y.o.u. .w.e.r.
:00130  65 00 20 00 62 00 6F 00 72 00 6E 00 3F 00 22 00 e. .b.o.r.n.?.".
:00140  2C 00 22 00 61 00 6E 00 73 00 77 00 65 00 72 00 ,.".a.n.s.w.e.r.
:00150  22 00 3A 00 22 00 5F 00 73 00 33 00 63 00 72 00 ".:."._.s.3.c.r.
:00160  33 00 74 00 73 00 5F 00 22 00 7D 00 2C 00 7B 00 3.t.s._.".}.,.{.
:00170  22 00 71 00 75 00 65 00 73 00 74 00 69 00 6F 00 ".q.u.e.s.t.i.o.
:00180  6E 00 22 00 3A 00 22 00 57 00 68 00 61 00 74 00 n.".:.".W.h.a.t.
:00190  20 00 77 00 61 00 73 00 20 00 79 00 6F 00 75 00  .w.a.s. .y.o.u.
:001A0  72 00 20 00 63 00 68 00 69 00 6C 00 64 00 68 00 r. .c.h.i.l.d.h.
:001B0  6F 00 6F 00 64 00 20 00 6E 00 69 00 63 00 6B 00 o.o.d. .n.i.c.k.
:001C0  6E 00 61 00 6D 00 65 00 3F 00 22 00 2C 00 22 00 n.a.m.e.?.".,.".
:001D0  61 00 6E 00 73 00 77 00 65 00 72 00 22 00 3A 00 a.n.s.w.e.r.".:.
:001E0  22 00 72 00 33 00 67 00 31 00 73 00 74 00 72 00 ".r.3.g.1.s.t.r.
:001F0  79 00 7D 00 22 00 7D 00 5D 00 7D 00             y.}.".}.].}.
```

The flag is in the answer: GCTF{p3rs0nal_s3cr3ts_r3g1stry}!

Find Me 1



Challenge Overview

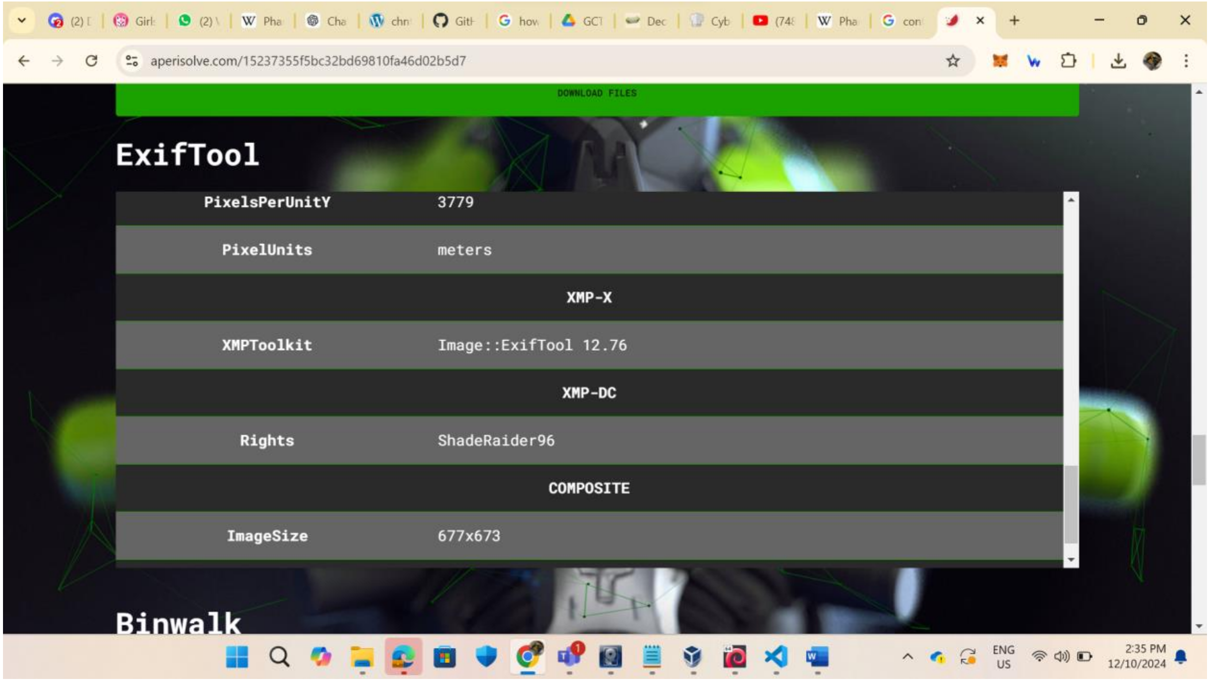
In this OSINT challenge, I was tasked with identifying the real name of a high-profile hacker who leaves images after compromising systems. My goal was to analyze the given image and find any clues or "fingerprints" the hacker may have left behind, ultimately revealing their true identity.

Step 1: Image Analysis with Aperisolve

I started by analyzing the provided image using Aperisolve, an online tool used for in-depth analysis of image files. Aperisolve can extract hidden data, metadata, and other useful information from image files, making it an essential tool for this challenge. Once I uploaded the image to Aperisolve, I reviewed the output from various analysis tools. One key result came from the ExifTool command, which extracts metadata embedded within the image.

Step 2: Metadata Analysis

In the ExifTool output, I found an important clue under the metadata field labeled "Rights". The entry revealed a username:



Rights: ShadeRaider96

Step 3: Investigating the Username

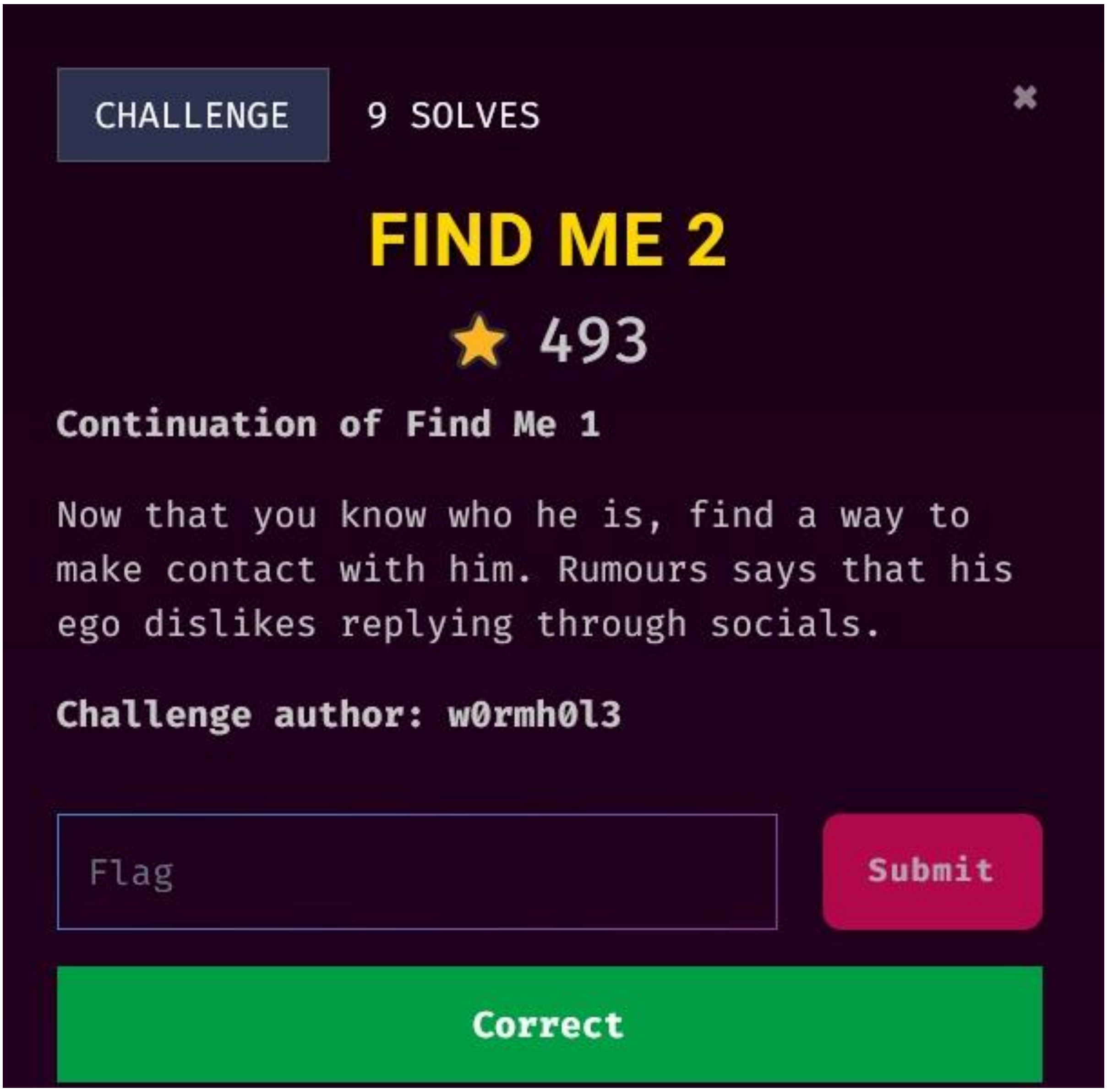
With the username ShadeRaider96, I began searching for traces of this alias across different online platforms. Through my investigation, I discovered an account associated with this username on Twitter (X).

The Twitter (X) profile for ShadeRaider96 contained public information, including posts and interactions. Most importantly, it also revealed the hacker's real name in the bio section.

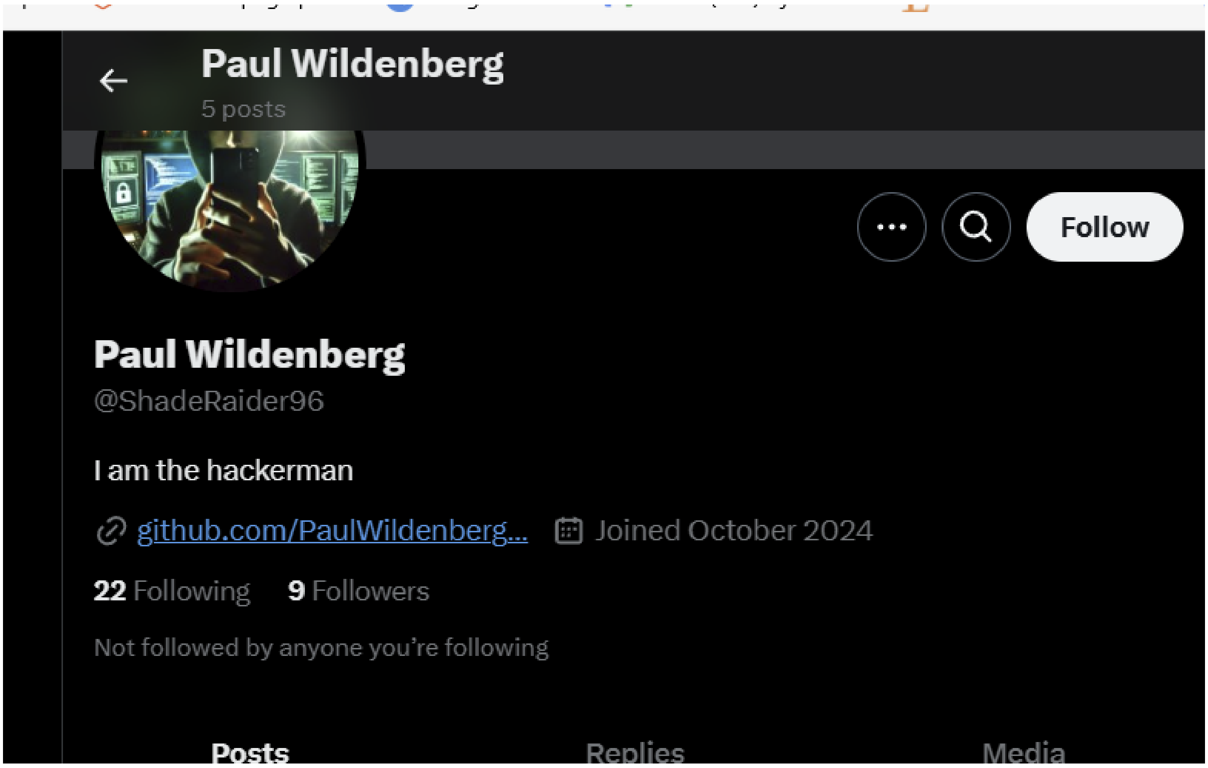
Step 4: Identifying the Real Name

After cross-referencing the information from the Twitter (X) profile, I was able to confirm the hacker’s real name. This successfully completed the challenge.

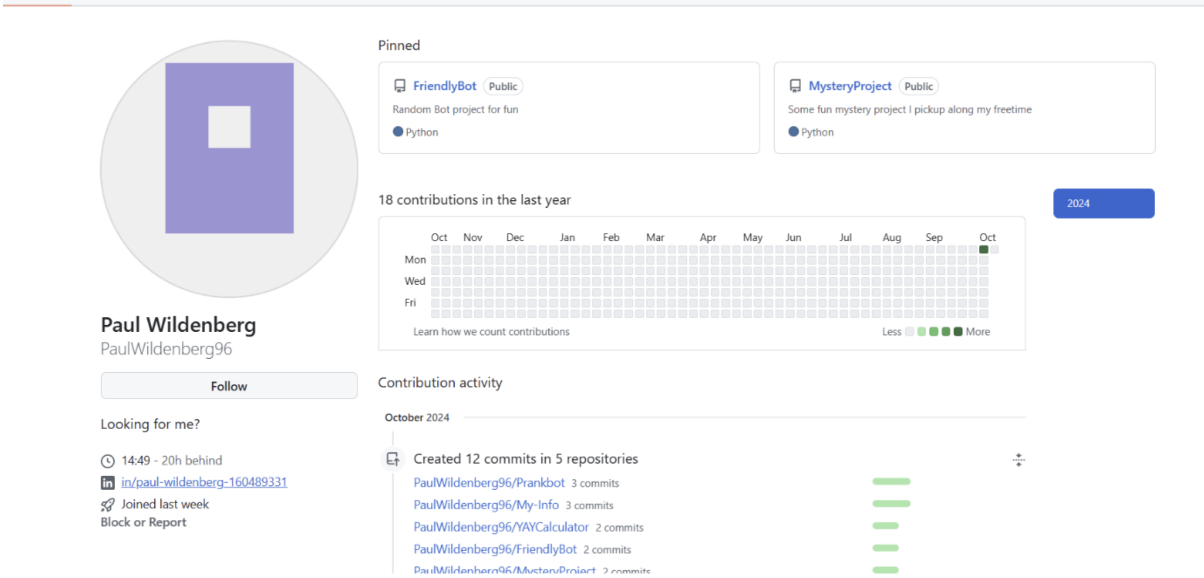
Flag: gctf{Paul_Wildenberg}!



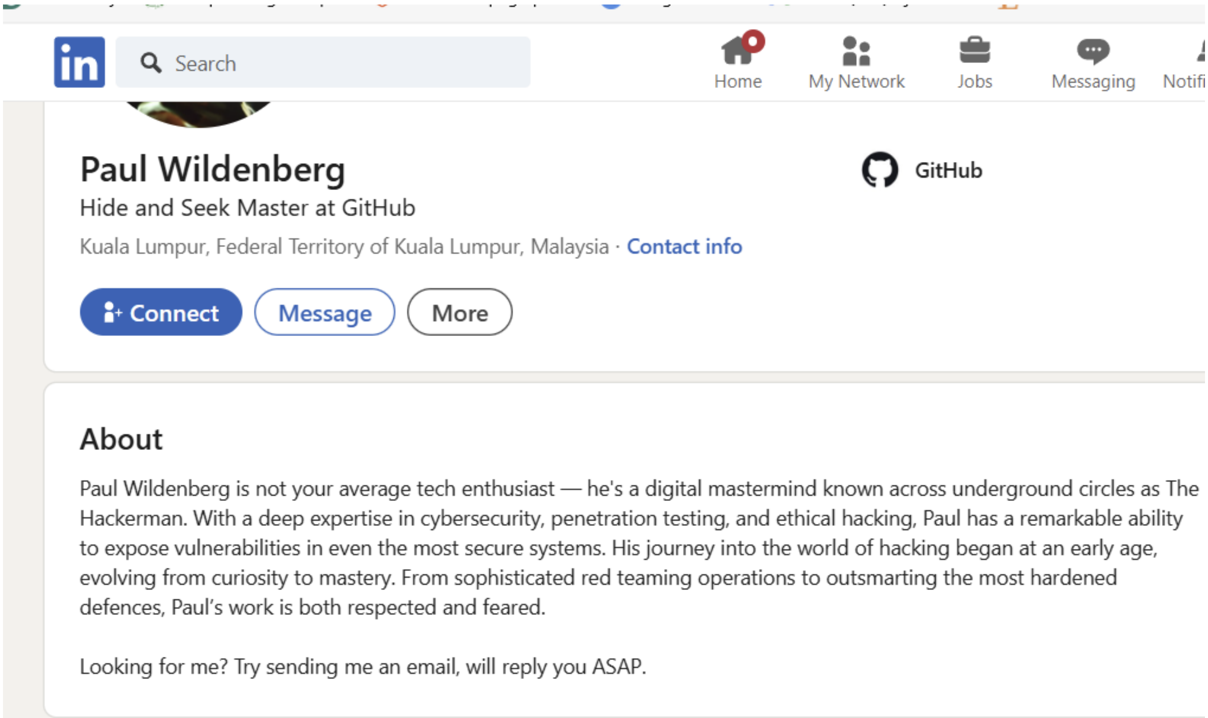
The question said find a way to contact with him so I looking at his twitter and found his Github link in bio.



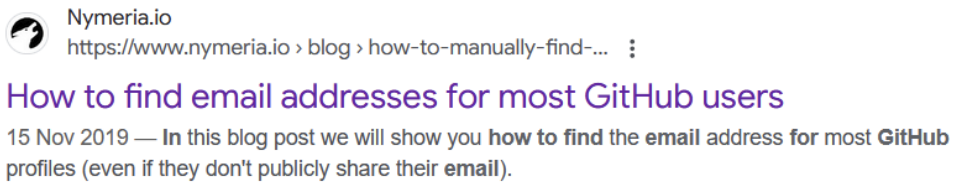
And at his github, I found his Linkedln link.



At the About in Linkedln, he said try send email to contact him.

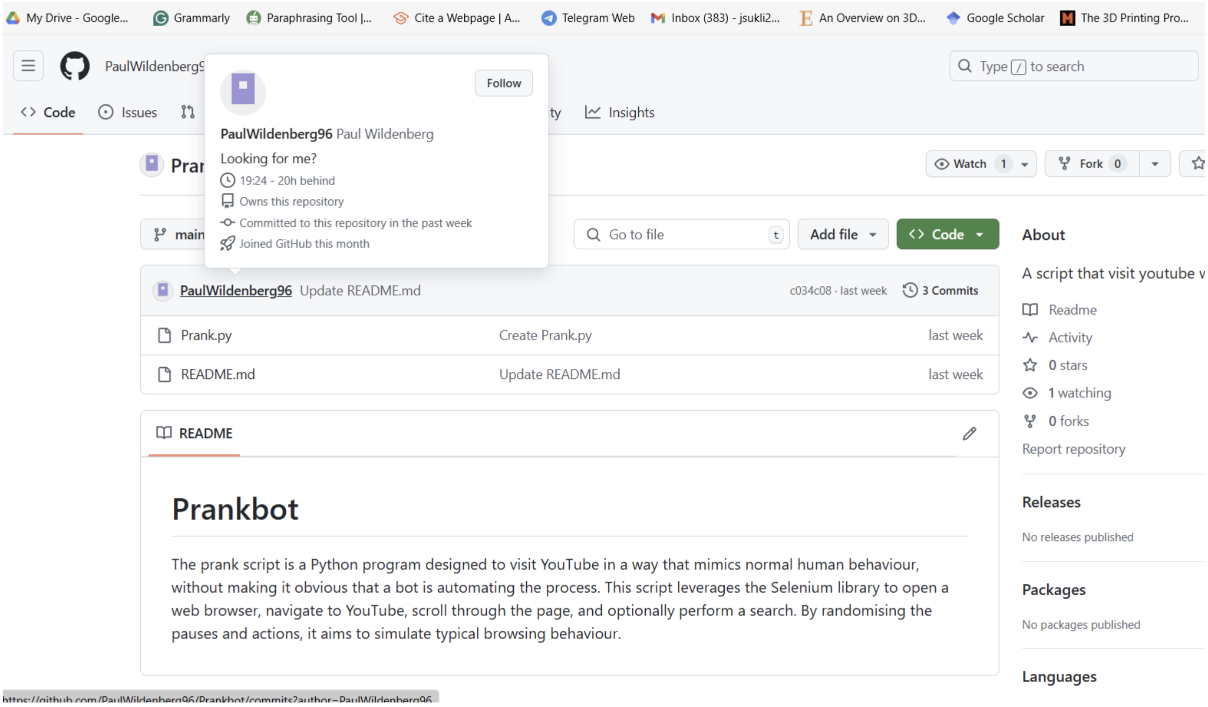


So, what I need to do no is found his email according to the current information we have. Therefore, I try search is there any way can find a person email through the GitHub. And I found it!

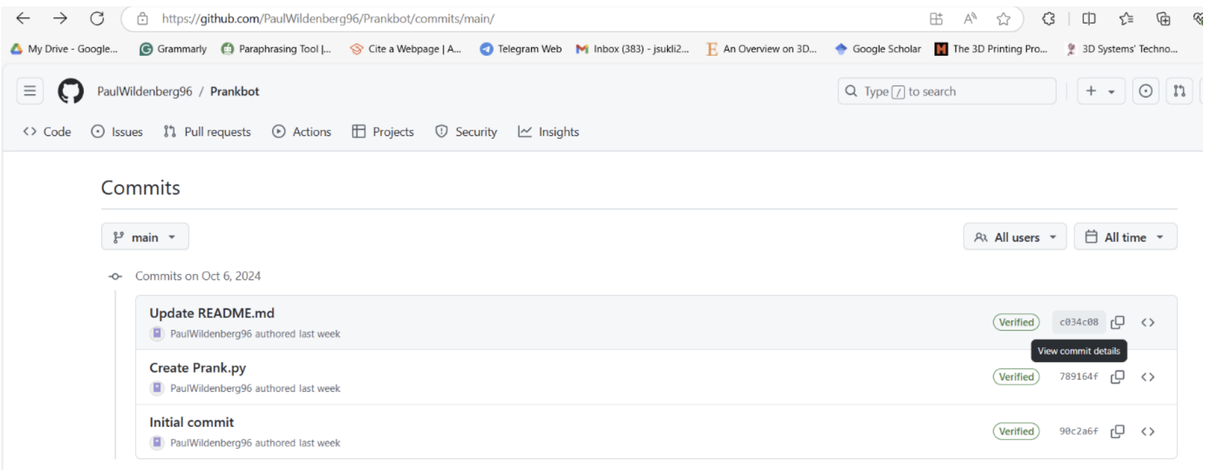


Link: https://www.nymeria.io/blog/how-to-manually-find-email-addresses-for-github-users

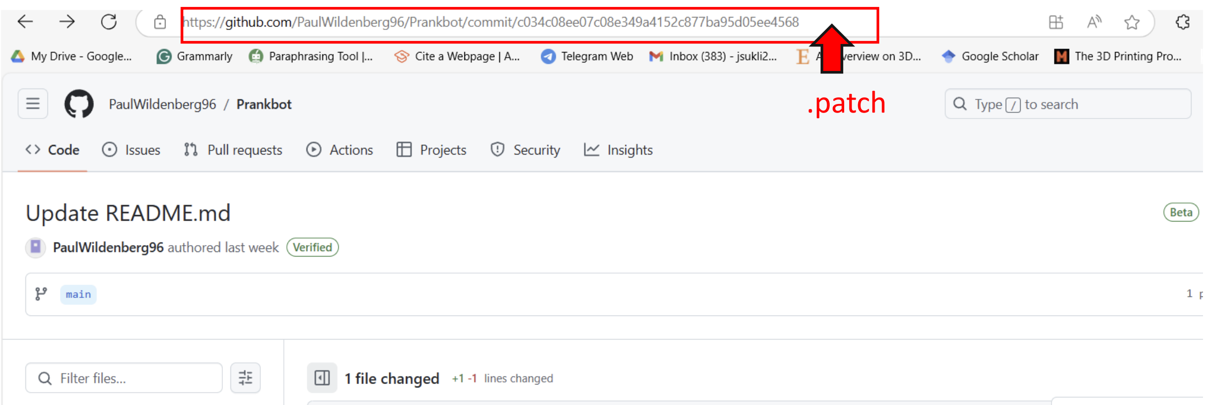
Firstly, go to any of his non-forked repository:



Find the commit by the user:



Once locate a promising commit click on the commit ID on the right hand side to view the actual commit.

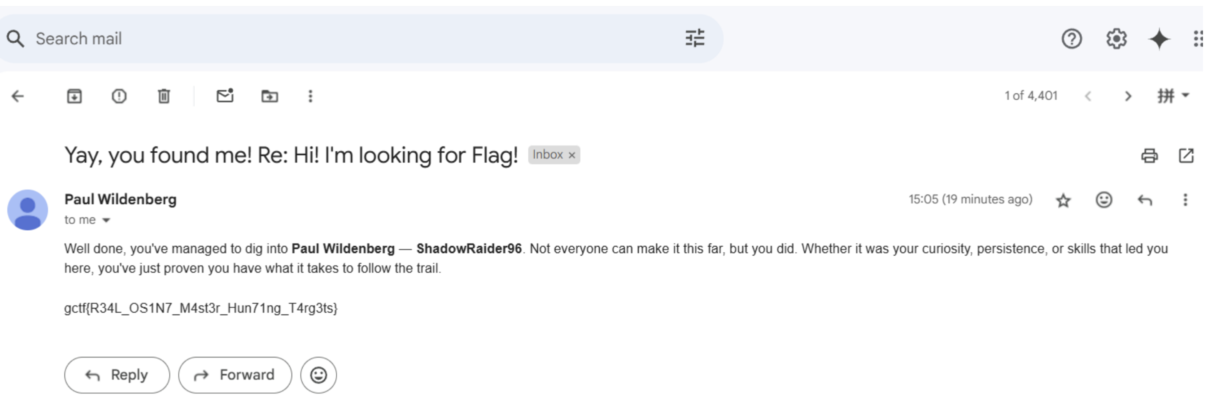


Convert to the patch view to locate the email address

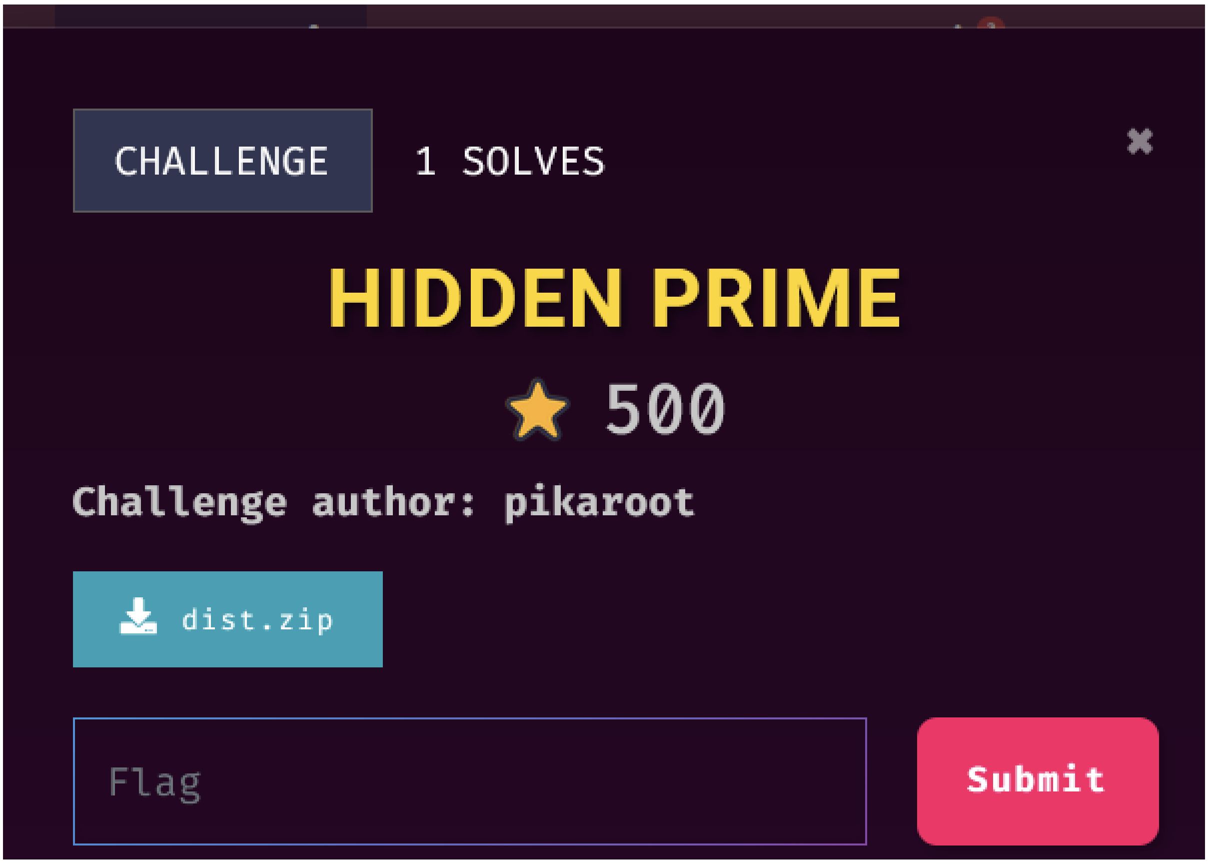
We found his email.

Next step is contact him with this email:

Finally, he reply to me with the flag.



Cryto – Hidden Prime




```python
from Crypto.Util.number import getPrime, bytes_to_long as b2l
from secret import flag
from secrets import token_bytes as tb

E = b2l(tb(100))

flag1 = b2l(flag[:16])
flag2 = b2l(flag[16:])

p, q, x = [getPrime(1024) for _ in range(3)]

n1, n2 = p*x, q*x
e = 0x10001

c1 = pow(flag1,e,n1)
c2 = pow(flag2,e,n2)

with open('out.txt','w') as f:
    f.write(f'n1: {n1}\n')
    f.write(f'c1: {c1}\n')
    f.write(f'c2: {c2}\n')
    f.write(f'hint: {n1*E+n2}')
```

The code given shows this challenge involves RSA encryption, where two parts of the flag (flag1 and flag2) are encrypted with the same e (public exponent) but different moduli (n1 and n2). Both moduli share a common factor x, making it possible to recover the flag by calculating that common factor and then using it to decrypt the ciphertexts.

Here is the given data in txt file: n1:
16982669419096392392726727892050630477471293600977138796406099616665036623380989094730047551830994334269527276925787718384952254916441196383736133996608207077306749354344692878295763558129849526110980328644660776223007727051478355569269450390557368191927431744201092701111907887820745690063322735854307381928543241269677474817329340405860347939823024660558611217766738923349260204407304144831628141110660724506571627942264953456713735245944960401365486685360888103477793378852623490489212290916224502642500923177984387482447337503507993198280826469940498418325092540693018383225176933294589423818557003478593666138697

c1:
4121897285069672533289462062837762004836963567074263903595433058484802602349420025458286725130254393698465925106362352790531792390417320779205336425907354580916980013972134132225027151508104959097958202738234027357398970317511041268200245307823496263943937049513830485937959512303695585308492749505368988090959924907535986303839119322061827441718333056405667715128754022201952341720207239965165829486335361699151726589158060020849304849737750347050023270450918492070687660219998198421071351597415036748441719018618708570784374235712221276464730709070944806614851257142714515743147897418599637560585882481622156452959

c2:
6437821642787048285163037822842363188819901084611533267677491804289250873674011552997990351873524948588691988224051588225310382727793022174803162247287055896165700320188939646960435911653273386866181520767005407067142384566295285284409658719936568559027184030692259317462434147980638530966899887488267278717435794733825883946944573679775078700404719734449642281503939290169468772810793089359840918513732899241358175722593042407571423162728694879449768012434426574883630615200770789587320509391523137583957379670529633023089450872351468380219196139130502214215474066443047220517620744431351503950851665477608448311216

hint:
3530156071052345579423867199489219685176129211352797109201348427077802510560555311037083102721772069652561879832292155889860364312493886085097985624220508184371060437167735952689307599

There is:

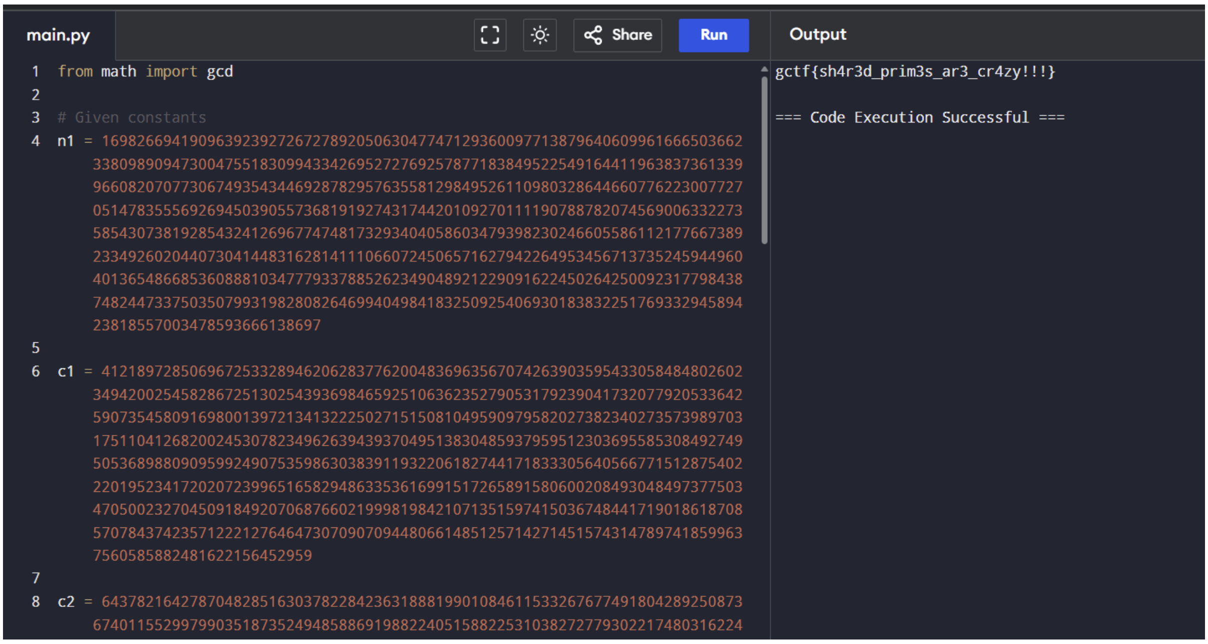
2 ciphertexts: c1 and c2

2 moduli: n1 and n2

A "hint" that is related to both moduli (n1 and n2) and some secret number E.

To recover flag1 and flag2 by decrypting c1 and c2. To do this, I need the private keys for both moduli n1 and n2.

Below is the python script I used to find the flag.

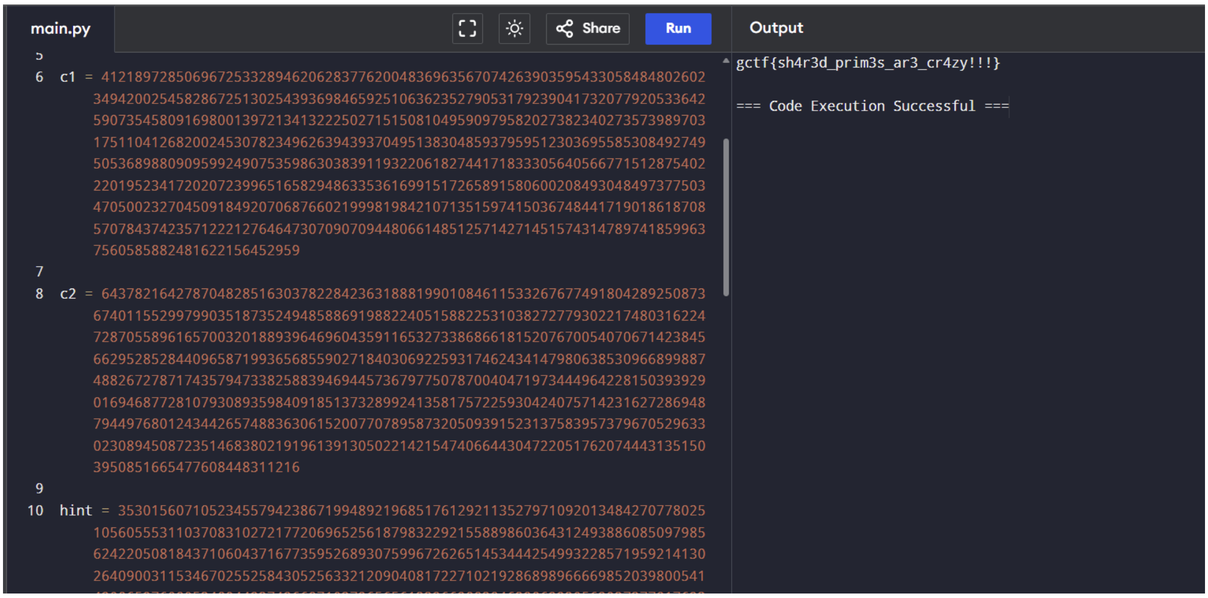


main.py

```python
from math import gcd

# Given constants
n1 = 16982669419096392392726727892050630477471293600977138796406099616665036623380989094730047551830994334269527276925787718384952254916441196383736133996608207077306749354344692878295763558129849526110980328644660776223007727051478355569269450390557368191927431744201092701111907887820745690063322735854307381928543241269677474817329340405860347939823024660558611217766738923349260204407304144831628141110660724506571627942264953456713735245944960401365486685360888103477793378852623490489212290916224502642500923177984387482447337503507993198280826469940498418325092540693018383225176933294589423818557003478593666138697

c1 = 412189728506967253328946206283776200483696356707426390359543305848480260234942002545828672513025439369846592510636235279053179239041732077920533642590735458091698001397213413222502715150810495909795820273823402735739897031751104126820024530782349626394393704951383048593795951230369558530849274950536898809095992490753598630383911932206182744171833305640566771512875402220195234172020723996516582948633536169915172658915806002084930484973775034705002327045091849207068766021999819842107135159741503674844171901861870857078437423571222127646473070907094480661485125714271451574314789741859963756058588248162215645295

8

c2 = 643782164278704828516303782284236318881990108461153326767749180428925087367401155299799035187352494858869198822405158822531038272779302217480316224
```

Output

```
gctf{sh4r3d_prim3s_ar3_cr4zy!!!}

=== Code Execution Successful ===
```



main.py


```python
c1 = 412189728506967253328946206283776200483696356707426390359543305848480260234942002545828672513025439369846592510636235279053179239041732077920533642590735458091698001397213413222502715150810495909795820273823402735739897031751104126820024530782349626394393704951383048593795951230369558530849274950536898809095992490753598630383911932206182744171833305640566771512875402220195234172020723996516582948633536169915172658915806002084930484973775034705002327045091849207068766021999819842107135159741503674844171901861870857078437423571222127646473070907094480661485125714271451574314789741859963756058588248162215645295
```


```python
c2 = 6437821642787048285163037822842363188819901084611533267677491804289250873674011552997990351873524948588691988224051588225310382727793022174803162247287055896165700320188939646960435911653273386866181520767005407067142384566295285284409658719936568559027184030692259317462434147980638530966899887488267278717435794733825883946944573679775078700404719734449642281503939290169468772810793089359840918513732899241358175722593042407571423162728694879449768012434426574883630615200770789587320509391523137583957379670529633023089450872351468380219196139130502214215474066443047220517620744431351503950851665477608448311216

hint = 353015607105234557942386719948921968517612921135279710920134842707780251056055531103708310272177206965256187983229215588986036431249388608509798562422050818437106043716773595268930759967262651453444254993228571959214130264090031153467025525843052563321209040817227102192868989666698520398005410
```

Output

```
gctf{sh4r3d_prim3s_ar3_cr4zy!!!}

=== Code Execution Successful ===
```

main.py

```python

hint = 35301560710523455794238671994892196851761292113527971092013484270778025105605553110370831027217720696525618798322921558898603643124938860850979856242205081843710604371677359526893075996726265145344425499322857195921413026409003115346702552584305256332120904081722710219286898966669852039800541420865276089534894423742668710372656561332662883846328622305629373778176234869216531193667540087665660313206120496373636873424680809698873223692668025586257096871543752086660200691342766826024186847510790006449674795802360619469005525909556211150532606337154280808386292483928324134821856188816129499628525445765513937907911882139405952971262813750454036478860704862854764846371687427901676018611325009792360072676788761698564467836032725211236959447433260416457739222282283772472685346578069287310114655326355070832373471840804207640483749541597385759366927740974

# Step 1: Calculate the common factor x
E = hint // (n1 % hint)
n2 = hint - (n1 * E)
x = gcd(n1, n2)

# Step 2: Retrieve the prime factors p and q
p = n1 // x
q = n2 // x

# Step 3: Calculate φ(n) = (p-1)(q-1)
phi_n1 = (p - 1) * (x - 1)  # Corrected to use x instead of q
```

Output

```
gctf{sh4r3d_prim3s_ar3_cr4zy!!!}

=== Code Execution Successful ===
```

main.py

```python
# Step 1: Calculate the common factor x
E = hint // (n1 % hint)
n2 = hint - (n1 * E)
x = gcd(n1, n2)

# Step 2: Retrieve the prime factors p and q
p = n1 // x
q = n2 // x

# Step 3: Calculate φ(n) = (p-1)(q-1)
phi_n1 = (p - 1) * (x - 1)  # Corrected to use x instead of q

# Step 4: Compute the private key d using the modular inverse
e = 0x10001
d1 = pow(e, -1, phi_n1)

# Step 5: Decrypt the ciphertexts
flag1 = pow(c1, d1, n1)

# We need to calculate φ(n2) and d2 as well
phi_n2 = (q - 1) * (x - 1)  # Calculate φ(n2)
d2 = pow(e, -1, phi_n2)  # Compute private key for n2

flag2 = pow(c2, d2, n2)  # Decrypt c2 with d2
```

Output

```
gctf{sh4r3d_prim3s_ar3_cr4zy!!!}

=== Code Execution Successful ===
```

main.py

```python
phi_n2 = (q - 1) * (x - 1)  # Calculate φ(n2)
d2 = pow(e, -1, phi_n2)  # Compute private key for n2

flag2 = pow(c2, d2, n2)  # Decrypt c2 with d2

# Convert long integers back to bytes
def long_to_bytes(n):
    """Convert a long integer to a byte string."""
    if n == 0:
        return b''
    byte_arr = bytearray()
    while n:
        byte_arr.append(n & 0xff)
        n >>= 8
    return bytes(reversed(byte_arr))

flag1_bytes = long_to_bytes(flag1)
flag2_bytes = long_to_bytes(flag2)

# Combine both parts and attempt to decode
full_flag = flag1_bytes + flag2_bytes

# Handle potential decoding errors
try:
    decoded_flag = full_flag.decode('utf-8')
```

Output

```
gctf{sh4r3d_prim3s_ar3_cr4zy!!!}

=== Code Execution Successful ===
```

main.py

```python
    """Convert a long integer to a byte string."""
    if n == 0:
        return b''
    byte_arr = bytearray()
    while n:
        byte_arr.append(n & 0xff)
        n >>= 8
    return bytes(reversed(byte_arr))

flag1_bytes = long_to_bytes(flag1)
flag2_bytes = long_to_bytes(flag2)

# Combine both parts and attempt to decode
full_flag = flag1_bytes + flag2_bytes

# Handle potential decoding errors
try:
    decoded_flag = full_flag.decode('utf-8')
except UnicodeDecodeError:
    decoded_flag = full_flag.decode('latin-1')  # Try a different encoding

# Return the decrypted flag
print(decoded_flag)
```

Output

```
gctf{sh4r3d_prim3s_ar3_cr4zy!!!}

=== Code Execution Successful ===
```

Code Explaination

Libraries

from math import gcd

- The gcd function from the math library is used to calculate the greatest common divisor, which is crucial for factoring *n.*

Constant

n1 is the modulus for the RSA encryption, and c1 and c2 are the ciphertexts want to decrypt. The hint is used to derive additional information about the modulus.

Step 1: Calculate the Common Factor

E = hint // (n1 % hint)

n2 = hint - (n1 * E)

x = gcd(n1, n2)

Finding x: The variable E calculates how many times *n1* fits into hint. The value of n2 is derived from hint, and gcd is used to find the common factor *xx*x between n1 and n2. This is crucial because if *n1* and *n2* share a prime factor, it can help in factorization.

Step 2: Retrieve the Prime Factors

p = n1 // x

q = n2 // x

The values of *p* and *q* are calculated by dividing *n1* and *n2* by the common factor *x*. These are the prime factors used in the RSA encryption.

Step 3: Calculate φ(n)

```
phi_n1 = (p - 1) * (x - 1)

phi_n2 = (q - 1) * (x - 1)
```

The totient function *φ(n)* is calculated for the modulus *n1 & n2*. This function is essential for determining the private key in RSA.

Step 4: Compute the Private Key

```
d1 = pow(e, -1, phi_n1)  # Private key for n1

 d2 = pow(e, -1, phi_n2)  # Private key for n2
```

With φ(n1) and φ(n2), we can compute the private keys d1 and d2 using the modular inverse of e modulo φ(n1) and φ(n2)

Step 5: Decrypt the Ciphertexts

```
flag1 = pow(c1, d1, n1)

flag2 = pow(c2, d2, n2)
```

Now that we have the private keys d1 and d2, we can decrypt the ciphertexts c1 and c2 to recover the two parts of the flag (flag1 and flag2)

Converting Long Integers to Bytes

```
def long_to_bytes(n):

   if n == 0:

      return b''

   byte_arr = bytearray()

   while n:

      byte_arr.append(n & 0xff)

      n >>= 8

   return bytes(reversed(byte_arr))
```

This function converts the decrypted integer values back into a byte string. This step is necessary because RSA encrypts data as integers, and you need to interpret those integers as bytes to get back the original message.

Combining the Flags and Decoding

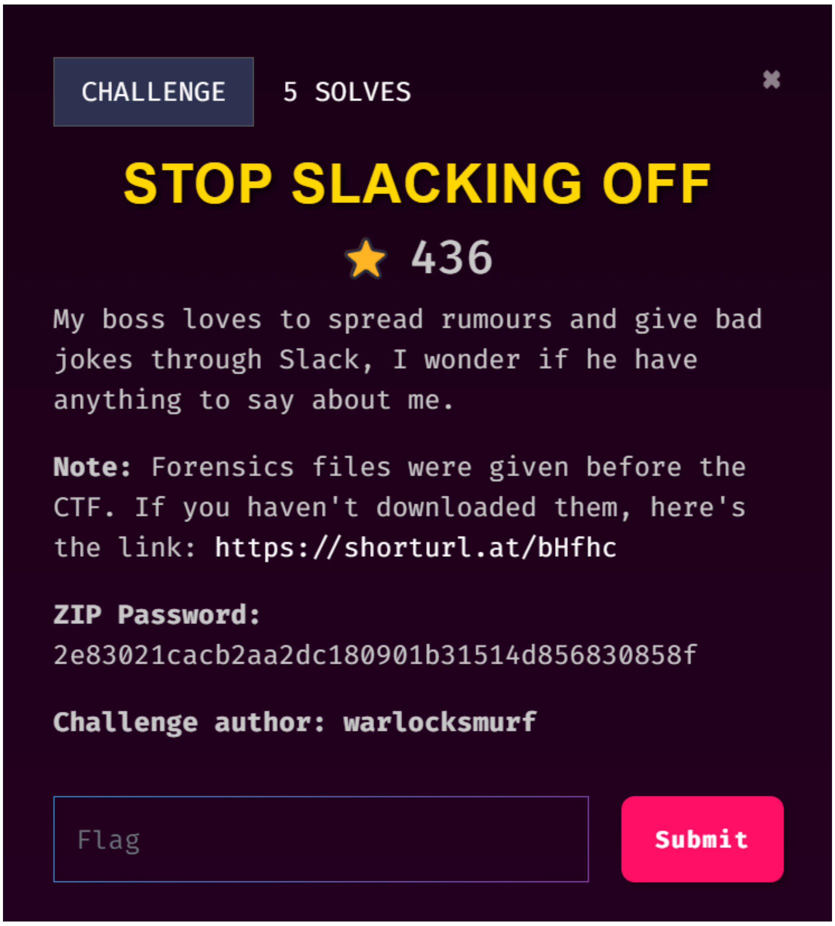
```
full_flag = flag1_bytes + flag2_bytes


try:
```

```
    decoded_flag = full_flag.decode('utf-8')
except UnicodeDecodeError:
    decoded_flag = full_flag.decode('latin-1')  # Try a different encoding


print(decoded_flag)
```

The decrypted byte sequences for both flags are combined. The script attempts to decode the combined bytes using UTF-8 encoding. If that fails, it falls back to latin-1, which can handle any byte value.

Flag: gctf{sh4r3d_prim3s_ar3_cr4zy!!!}

Stop Slacking Off



Challenge Overview

In the Stop Slacking Off forensic challenge, I was tasked with investigating a Slack workspace for any suspicious or humorous activity, especially rumours or jokes from the boss. The only data provided was from a disk image containing the path to Slack's user data folder. My goal was to find any conversations about rumours or jokes and potentially discover a secret code hidden in the Slack files.

Step 1: Exploring the Disk Image

The disk image contained a single path to Slack's application data:

C:\Users\GIC2024\AppData\Roaming\Slack

This folder had several subdirectories, including:

- Cache
- IndexedDB
- Local Storage
- Logs
- Storage

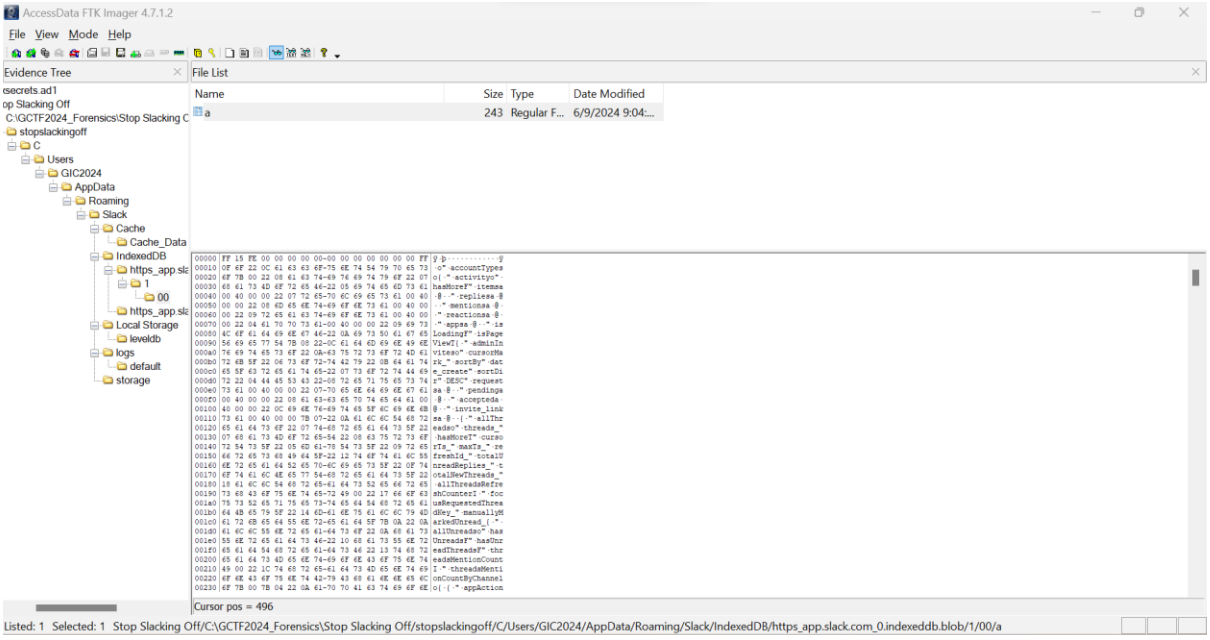
I focused on the **IndexedDB** folder because IndexedDB is a local database used by web applications like Slack to store structured data, including message history, channel information, and workspace interactions. This data could potentially contain the rumours or jokes mentioned in the challenge.

Step 2: Navigating the IndexedDB Directory

Within the **IndexedDB** folder, I navigated to:

C:\Users\GIC2024\AppData\Roaming\Slack\IndexedDB\https_app.slack.com_0.indexeddb.blob\1\00

Inside this directory, I found a regular file named **a**, which appeared to be a binary file containing local data from Slack. Given the importance of IndexedDB in storing key interactions, I opened this file in a text editor (Notepad) to analyse its content.

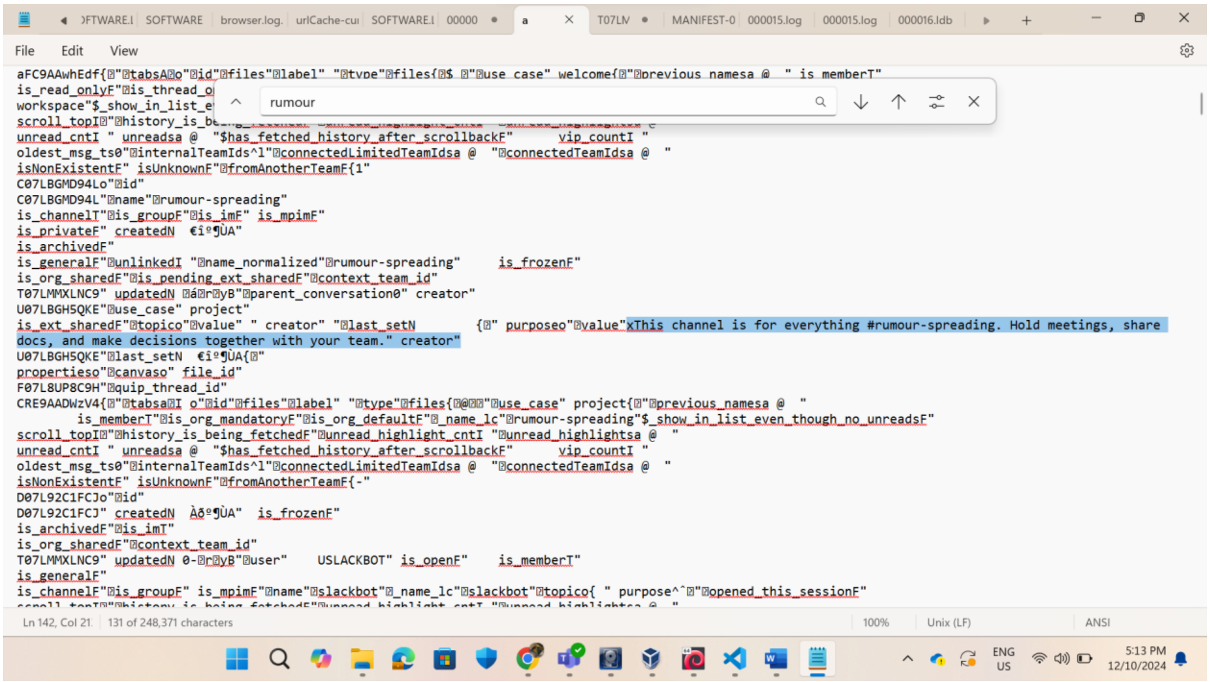


Step 3: Analyzing the IndexedDB File

I began examining the content of the file by searching for keywords relevant to the challenge description.

Searching for "rumour":

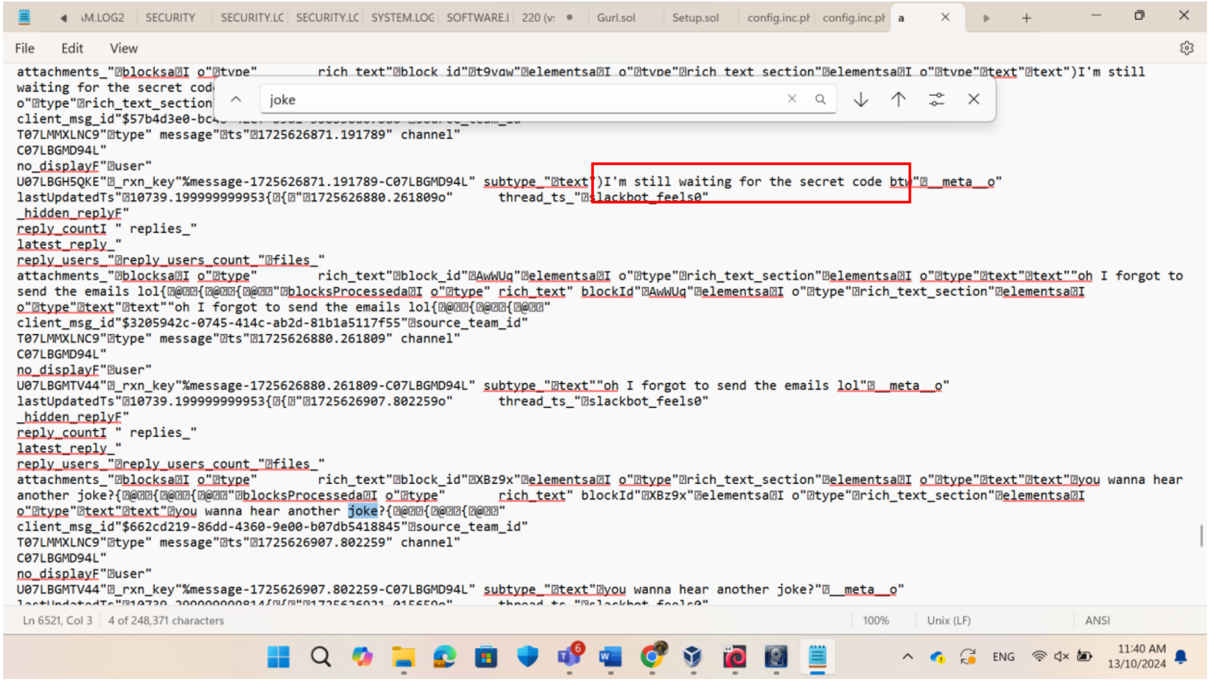
I searched for the word **rumour** and found the following message:



This indicated that there was a Slack channel specifically for spreading rumors, aligning with the challenge's suggestion that the boss spreads rumors. Finding this message confirmed that I was looking in the right file, and I decided to search for more keywords.

Searching for "joke":

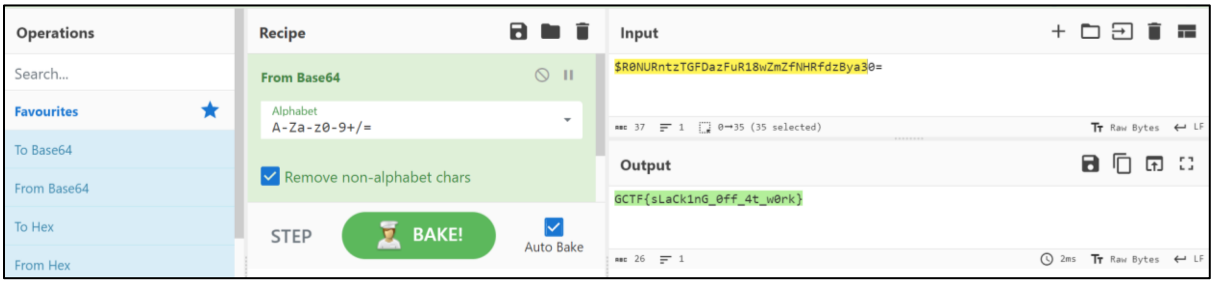
I searched for **joke** next and uncovered several communications between the team members, including a message referencing a **secret code**. This suggested that the secret code, possibly the flag, was hidden in the Slack conversations.



I followed the conversation trail and pieced together the context, revealing the code hidden within the chat logs.

Step 4: Extracting and Decoding the Secret Code

The secret code was encrypted using **Base64 encoding**. To decode the secret code, I used **CyberChef**, an online tool for various encryption and decryption tasks.



Flag: GCTF{sLaCk1nG_0ff_4t_w0rk}