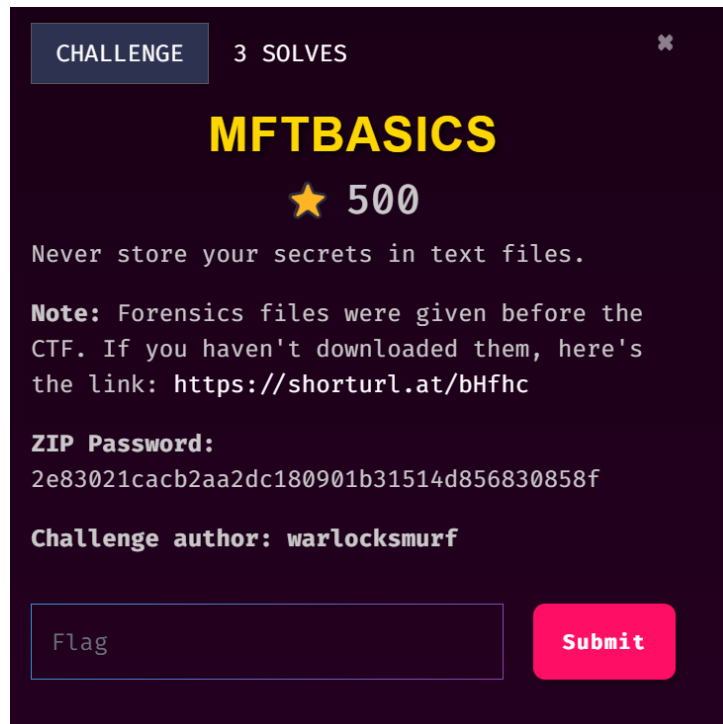


GCTF2024 Writeup

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.

```
(kali@kali)-[~/GCTF2024/MFTBasics]
$ strings \MFT| grep "flag"
#pragma class flags("forceupdate")
#pragma class flags(64)
#pragma class flags("forceupdate")
flag.txt
GIC2024\Desktop\flag.txt
flag.txt
GIC2024\Desktop\flag.txt

(kali@kali)-[~/GCTF2024/MFTBasics]
$
```

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:\GCTF2024_Forensics\MFTBasics\`$MFT" --dr --csv "C:\GCTF2024_Forensics\MFTBasics\Output"
MFTECmd version 1.2.2.1

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

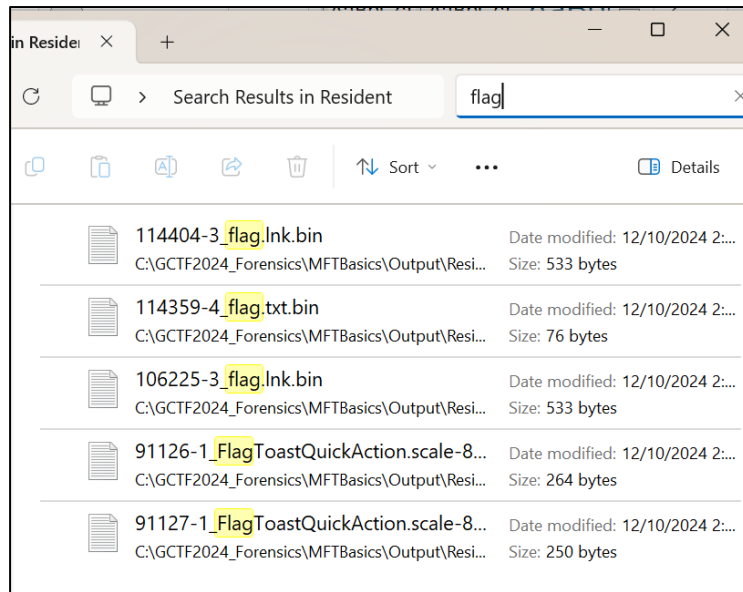
Command Line: -f C:\GCTF2024_Forensics\MFTBasics\`$MFT --dr --csv C:\GCTF2024_Forensics\MFTBasics\Output

Warning: Administrator privileges not found!

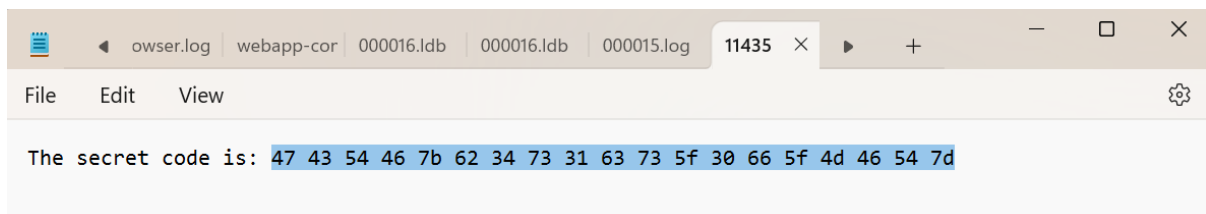
File type: Mft

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

C:\GCTF2024_Forensics\MFTBasics\`$MFT: FILE records found: 116,087 (Free records: 4,942) File size: 118.2MB
Path to C:\GCTF2024_Forensics\MFTBasics\Output doesn't exist. Creating...
  CSV output will be saved to C:\GCTF2024_Forensics\MFTBasics\Output\20241011181151_MFTECmd_`$MFT_Output.csv
  Resident data will be saved to C:\GCTF2024_Forensics\MFTBasics\Output\Resident
```

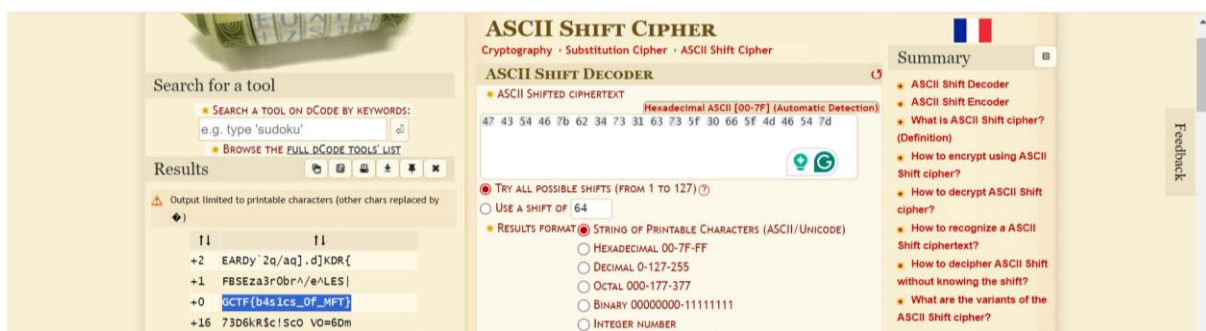


Well, after the command was done running, I went into the output directory, and Voilà – flag.txt.bin. I proceeded to open the flag.txt.bin in notepad and I got a **secret code**.



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_of_MFT}**!

Crypto -Warmup Salsa Sauce

CHALLENGE

9 SOLVES


✕

WARMUP SALSA SAUCE

★ 413

A skilled cybersecurity professional infiltrates a notorious drug cartel's communication network, managing to extract a series of encrypted messages that hold the key to unraveling their operations. The professional hands the data over to you, challenging them to break the code before it's too late.

Challenge author: pikaroot

 [dist.zip](#)

Submit

Encrypted flag:

fa1c26b66ad926ab75cd51524f05ec08f7f3160c74bec57f8aec3f7cace6fbb7370923e8c540673f657dada9e9540101d7f4dc0b6627f147fc47627a244c88b2ea6c3340

Encrypted text:

fa1c26b66ad926ab45cb05575b0ab90fcd1060d72bfba6f90aa076cbcf2f3aa311a13fc800638382570bcbee142001290f1d41d3761e315b91730663c2cd8e1fe7a25482ce0cd69745028635ef5dae54282f162e448fe

```

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:

```
1 from binascii import unhexlify
2
3 # The known plaintext (text variable from the script)
4 known_plaintext = b"We covered the drugs with our favourite salsa
  sauce. The stupid cops will not find it."
5
6 # The encrypted flag and text (hexadecimal format, provided in out.txt
  )
7 encrypted_flag_hex =
  "fa1c26b66ad926ab75cd51524f05ec08f7f3160c74bec57f8aec3f7cace6fbb73
  70923e8c540673f657dada9e9540101d7f4dc0b6627f147fc47627a244c88b2ea6
  c3340"
8 encrypted_text_hex =
  "fa1c26b66ad926ab45cb05575b0ab90fcdf1060d72bfa6f90aa076cbcf2f3aa3
  11a13fc800638382570bcbee142001290f1d41d3761e315b91730663c2cd8e1fe7
  a25482ce0cd69745028635ef5dae54282f162e448fec5f6b0e7d8ff85"
9
10 # Convert hex strings to bytes
11 encrypted_flag = unhexlify(encrypted_flag_hex)
12 encrypted_text = unhexlify(encrypted_text_hex)
13
14 # Extract the nonce (first 8 bytes of the ciphertext)
15 nonce = encrypted_text[:8]
16
17 # Extract the actual encrypted data (without the nonce)
18 encrypted_flag_data = encrypted_flag[8:]
19 encrypted_text_data = encrypted_text[8:]
20
21 # Derive the key stream by XORing the encrypted text with the known
  plaintext
22 keystream = bytes([enc_byte ^ plain_byte for enc_byte, plain_byte in
  zip(encrypted_text_data, known_plaintext)])
23
24 # Decrypt the flag by XORing the encrypted flag with the derived key
  stream
25 decrypted_flag = bytes([enc_byte ^ key_byte for enc_byte, key_byte in
  zip(encrypted_flag_data, keystream)])
26
27 # Output the decrypted flag
28 print(decrypted_flag.decode())
29
```

Code Explanation:

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

CHALLENGE

57 SOLVES



I FORGOT

★ 100

Here's a song that demonstrates real plagiarism. An artist sued the singers in this song and had won the lawsuit for over 5 million, ya then...uh....wait, I forgot in which courthouse they settled....yea uhm, welcome to osint! FIND THE COURTHOUSE!

Flag format: gctf{name_of_the_courthouse}

Challenge author: w0rmh0l3



Plagarism...

Flag

Submit

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

12:43

Vol 4G+ LTE2



Google



Robin Thicke Blurred Lines



All Images Videos Short videos News Shopping

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

Q

SUBSCRIBE

Top Stories

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

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

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

Mom Sends Daughter Grandparents' Door Sleepless Night (Excl

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

Q

Search Wikipedia

Search

Contents

hide

(Top)

Background

Complaint for declaratory relief

Plaintiff's disposition

Counter-claim

Motion for summary judgement

Holding

Trial

Appeal

Dissent

Subsequent developments

Commentary and implication

Article

Talk

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

Appeals

gctf{united_states_court_of_appeals_for_the_ninth_circuit}

Fox Secrets

CHALLENGE 6 SOLVES ✕

FOX SECRETS

★ 436

I might have accidentally place the flag as a password on Firefox, I hope Mr Mozarella is not mad at me ...

Note: Forensics files were given before the CTF. If you haven't downloaded them, here's the link: <https://shorturl.at/bHfhc>

ZIP Password:
2e83021cacb2aa2dc180901b31514d856830858f

Challenge author: warlocksmurf

Submit

Challenge Overview

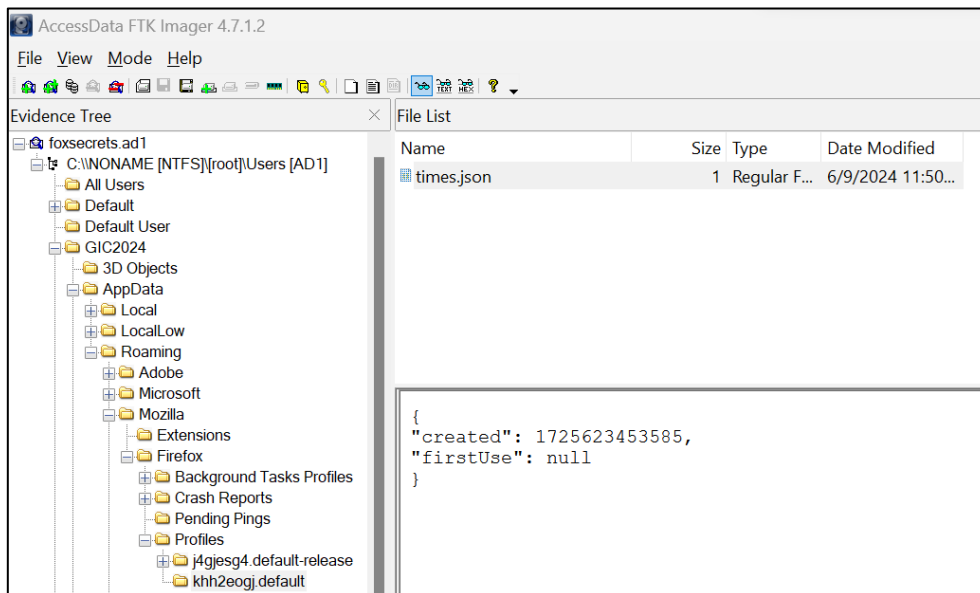
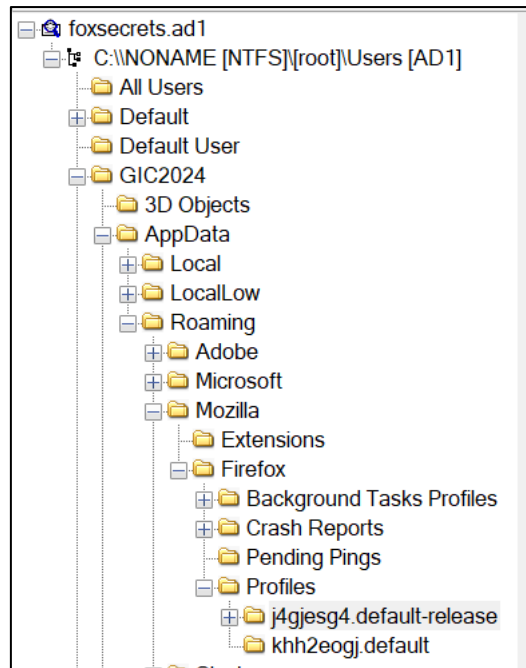
In the Fox Secret challenge, I was given a disk image that needed to be examined. The description stated that the flag might have been accidentally stored as a password in Firefox and the hint was "Mr. Mozarella", pointing toward Mozilla Firefox. My task is to extract the stored password, likely the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

Step 1: Initial Investigation of the Disk Image

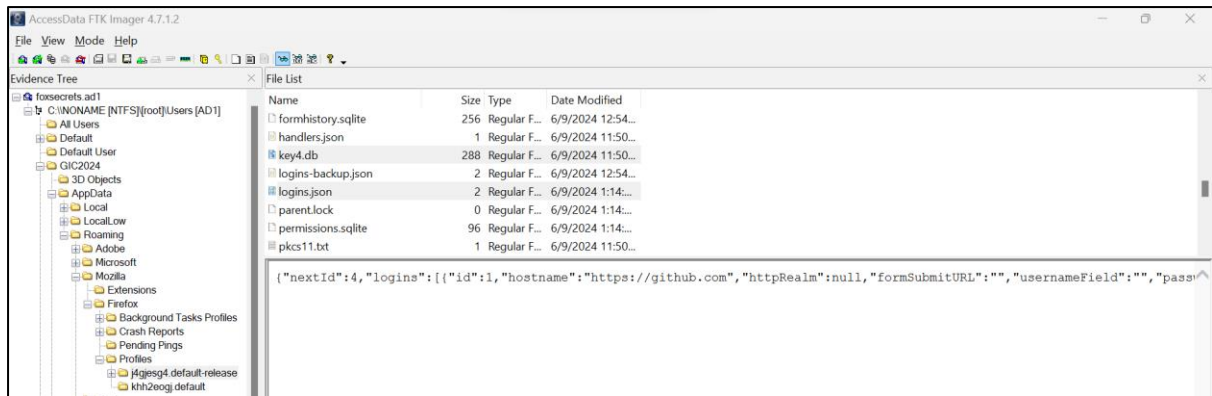
After obtaining the image of the disk, the first thing I did was to research about Firefox on where they store user information. I found out that the profile directory is a place where Firefox will store information related to the user, for instance: browsing history, bookmarks, saved passwords.

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 khk2eogj.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.

```
(kali@kali)-[~/./mozilla/firefox/6quv9d1x.default-esr]
$ cd /home/kali/GCTF2024/Fox\ Secrets/firefox_decrypt

(kali@kali)-[~/GCTF2024/Fox Secrets/firefox_decrypt]
$ python firefox_decrypt.py
Select the Mozilla profile you wish to decrypt
1 → dbgwa59r.default
2 → 6quv9d1x.default-esr
2

Website: https://github.com
Username: 'warlocksmurf'
Password: 'GCTF{m0zarella_'

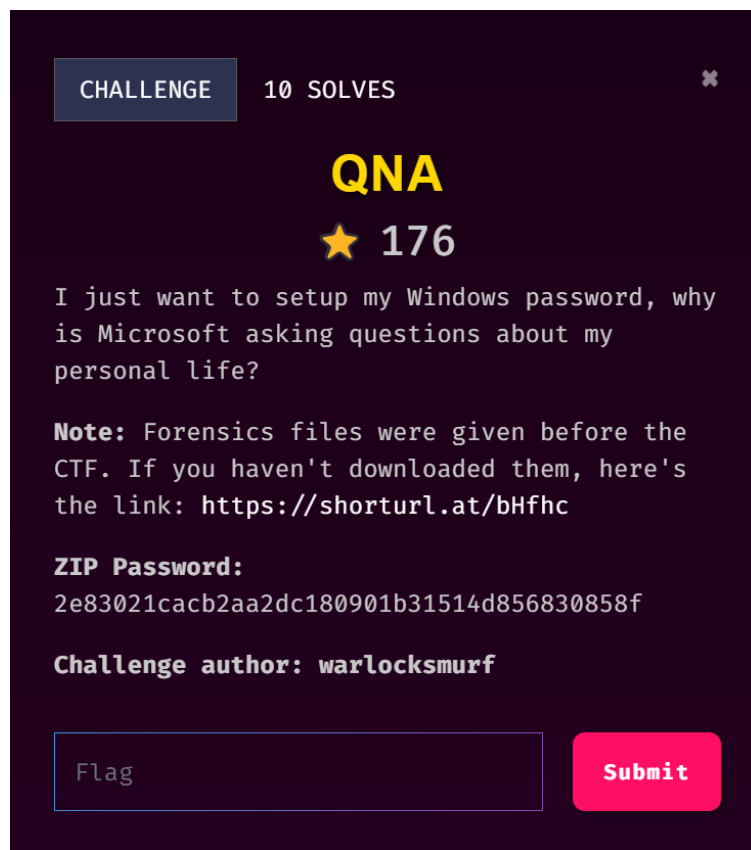
Website: https://www.rehack.xyz
Username: 'bluelobster'
Password: 'f1ref0x_p4ssw0rd}'

Website: https://ctftime.org
Username: 'pepperonilover'
Password: 'ilovecheese'

(kali@kali)-[~/GCTF2024/Fox Secrets/firefox_decrypt]
$
```

The flag is split into two parts. **GCTF{m0zarella_f1ref0x_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: 0x001020 * 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 [0x95b930c
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 [0x95b930c
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 [0x1fc]
: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...

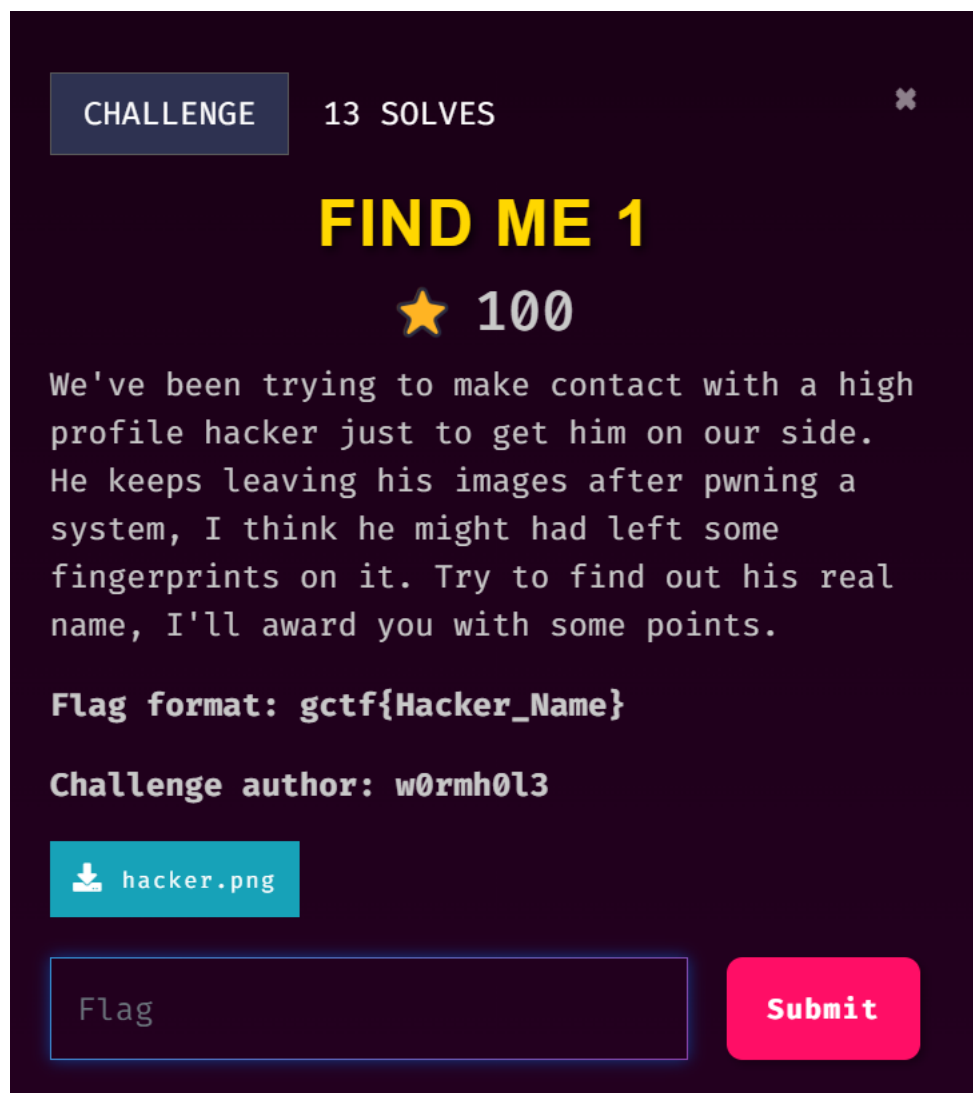
```

```

: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 68 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.i.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_r3glstry}!**



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.

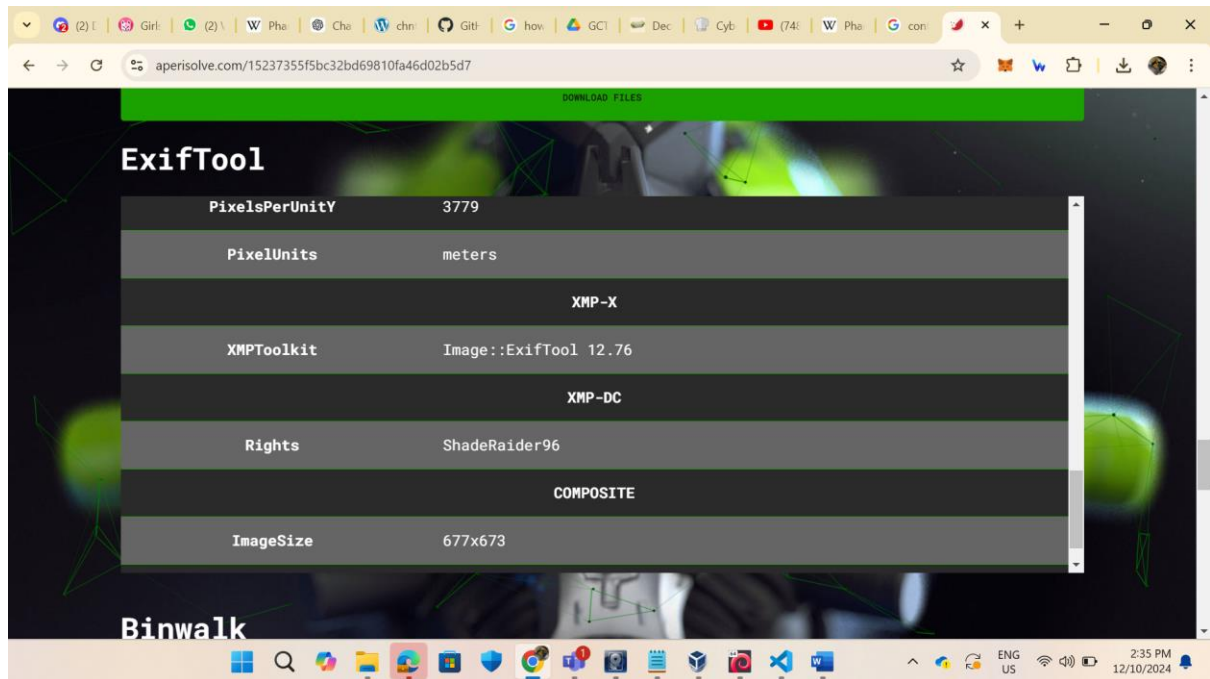
Step 1: Image Analysis with Aperisolve

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.

```
gctf{Paul_Wildenberg}
```

CHALLENGE

9 SOLVES



FIND ME 2

★ 493

Continuation of Find Me 1

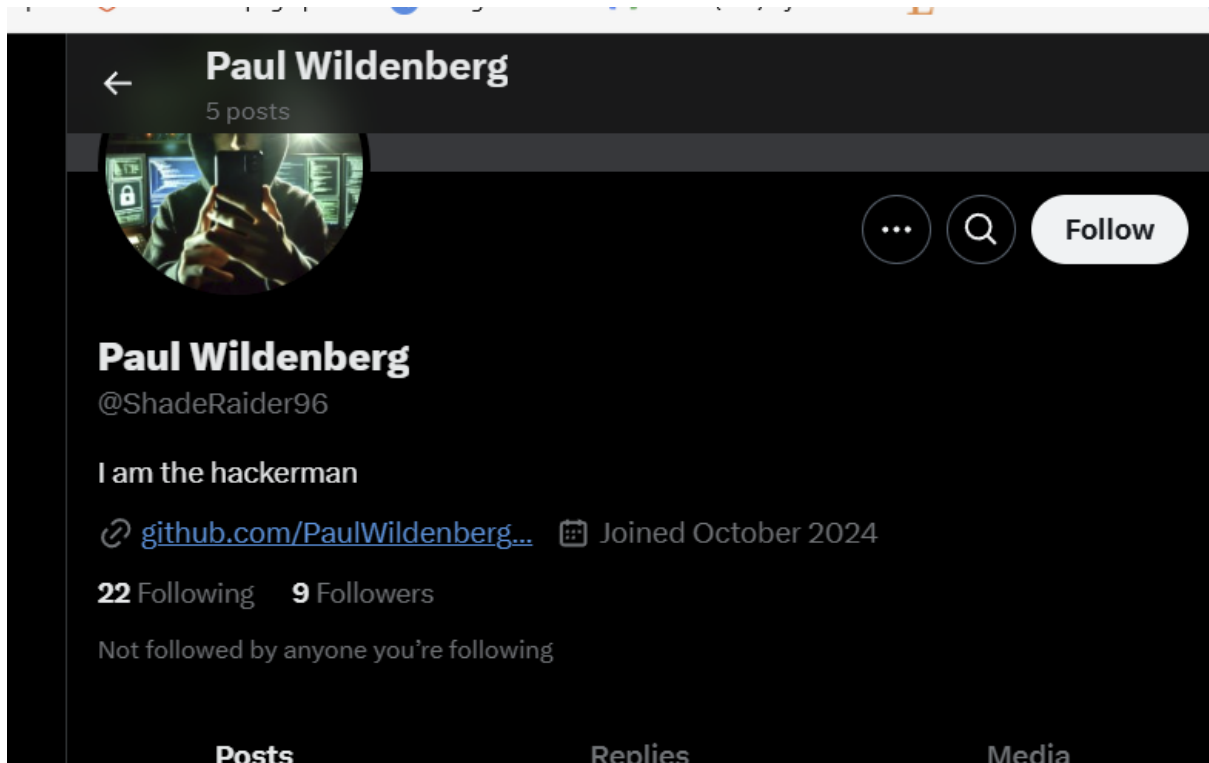
Now that you know who he is, find a way to make contact with him. Rumours says that his ego dislikes replying through socials.

Challenge author: w0rmh0l3

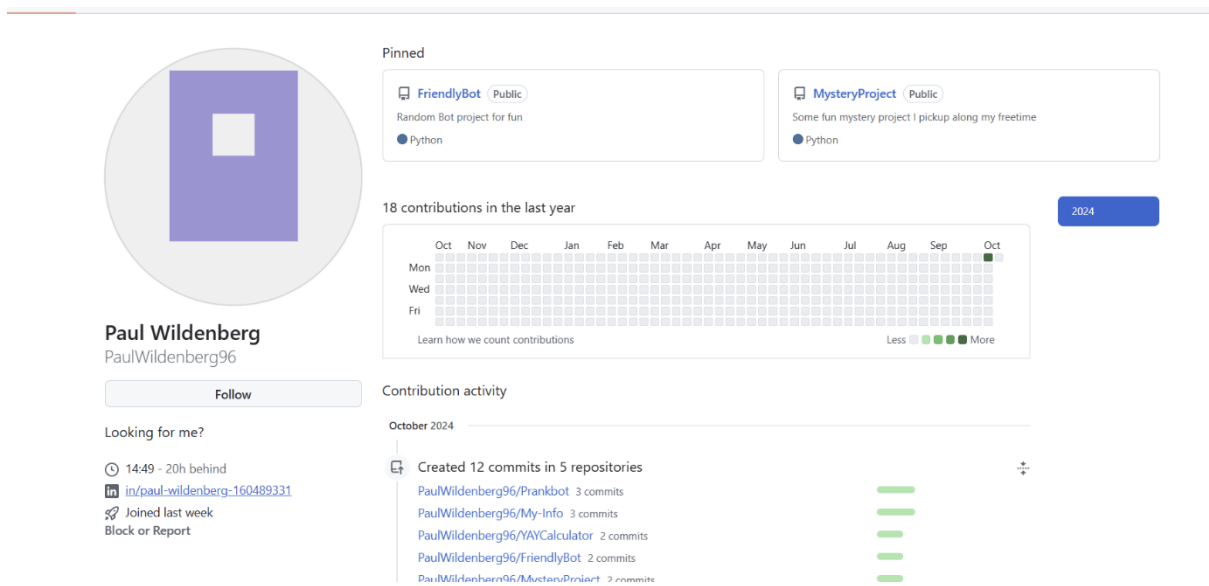
Submit

Correct

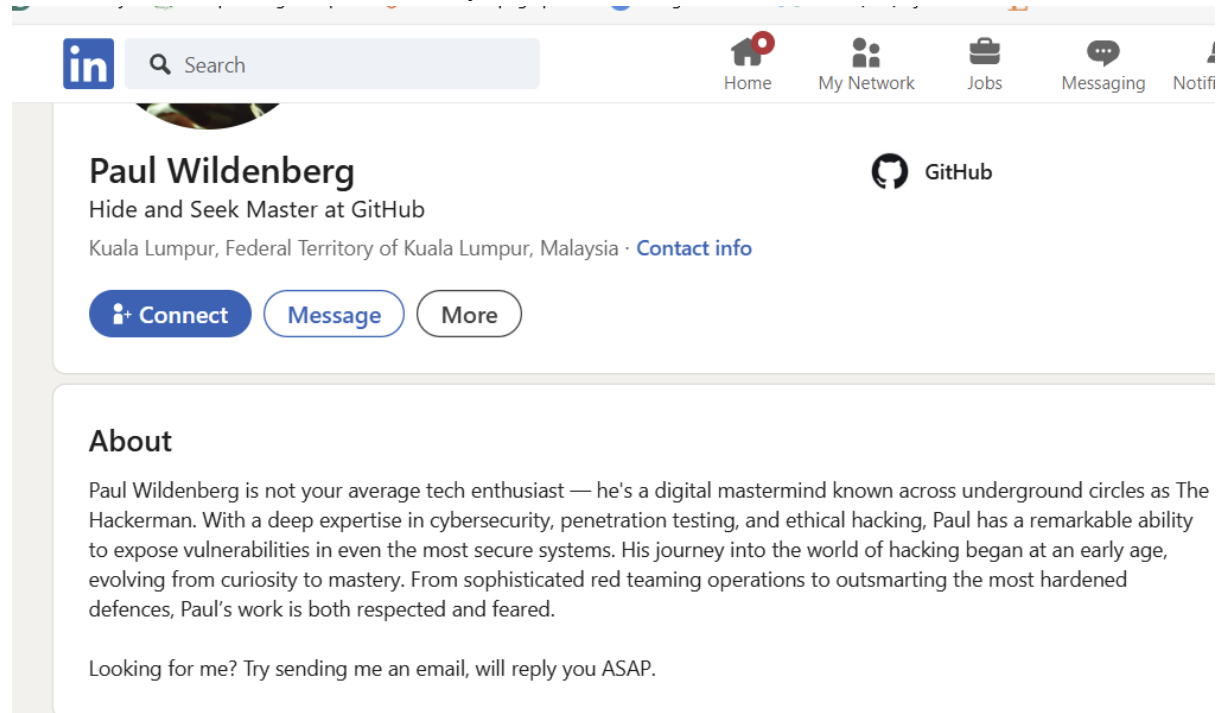
The question said find a way to contact with him so I looking at his twitter and found his Github link in bio.



And at his github, I found his LinkedIn link.

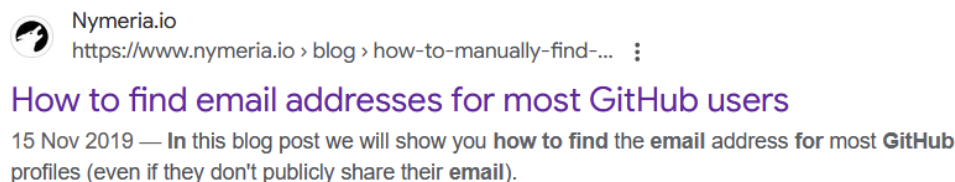


At the About in LinkedIn, he said try send email to contact him.



The screenshot shows the LinkedIn profile of Paul Wildenberg. The header includes the LinkedIn logo, a search bar, and navigation links for Home, My Network, Jobs, Messaging, and Notifications. The profile section displays his name, title 'Hide and Seek Master at GitHub', location 'Kuala Lumpur, Federal Territory of Kuala Lumpur, Malaysia', and a 'Contact info' link. Below this are buttons for 'Connect', 'Message', and 'More'. The 'About' section contains a detailed bio: 'Paul Wildenberg is not your average tech enthusiast — he's a digital mastermind known across underground circles as The Hackerman. With a deep expertise in cybersecurity, penetration testing, and ethical hacking, Paul has a remarkable ability to expose vulnerabilities in even the most secure systems. His journey into the world of hacking began at an early age, evolving from curiosity to mastery. From sophisticated red teaming operations to outsmarting the most hardened defences, Paul's work is both respected and feared.' It also includes a note: 'Looking for me? Try sending me an email, will reply you ASAP.'

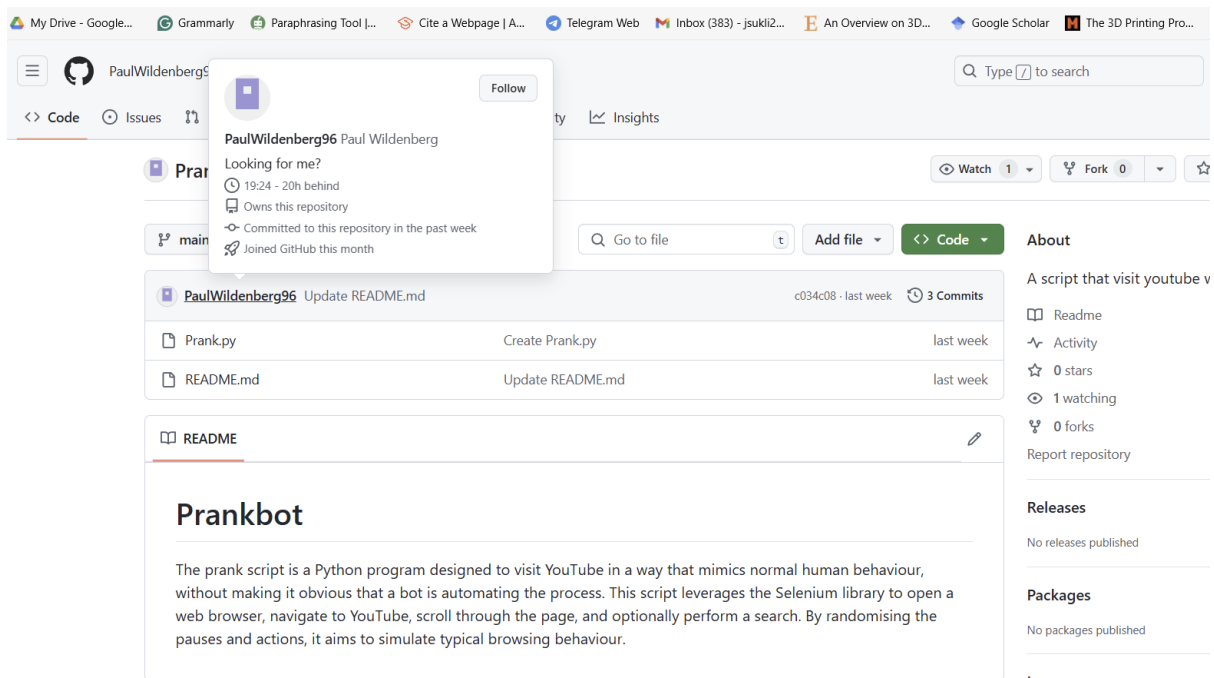
So, what I need to do now is find his email according to the current information we have. Therefore, I try search is there any way can find a person email through the GitHub. And I found it!



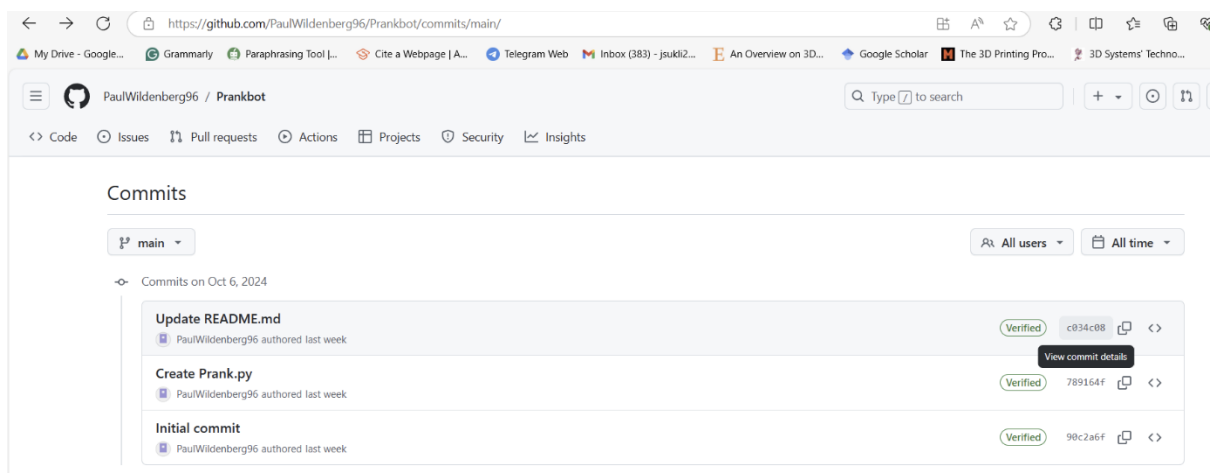
The screenshot shows a blog post from Nymeria.io. The breadcrumb trail is 'https://www.nymeria.io > blog > how-to-manually-find-...'. The title is 'How to find email addresses for most GitHub users'. The date is '15 Nov 2019'. The text reads: 'In this blog post we will show you how to find the email address for most GitHub profiles (even if they don't publicly share their email).'

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

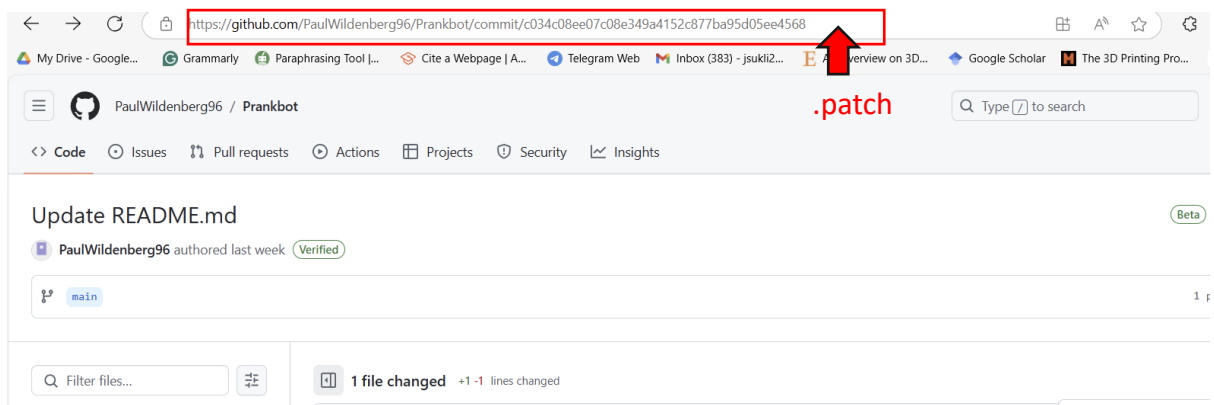
Firstly, go to any of his non-forked repository:



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

```
<img alt="GitHub commit patch view for PaulWildenberg96/Prankbot" data-bbox="117 110 878 355"/>
https://github.com/PaulWildenberg96/Prankbot/commit/c034c08ee07c08e349a4152c877ba95d05ee4568.patch

From c034c08ee07c08e349a4152c877ba95d05ee4568 Mon Sep 17 00:00:00 2001
From: PaulWildenberg96 <paulwildenberg96@gmail.com>
Date: Sun, 6 Oct 2024 02:26:30 +0800
Subject: [PATCH] Update README.md

---
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

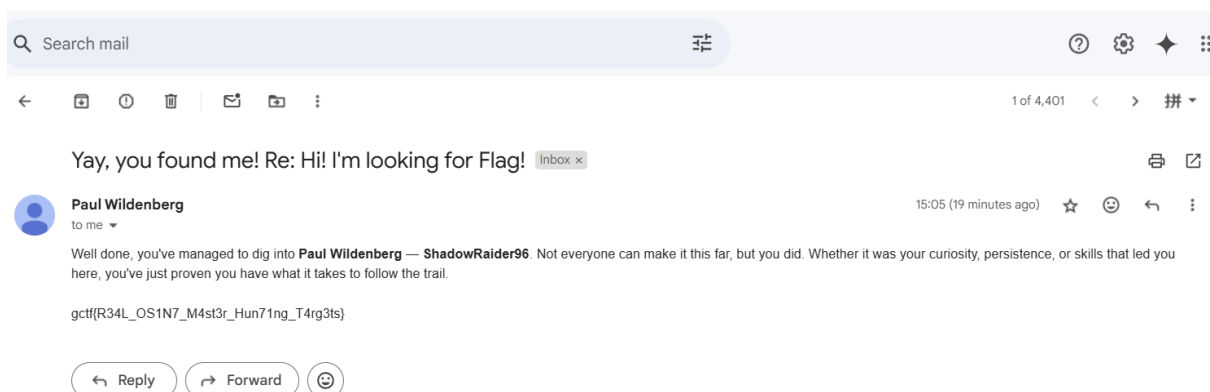
diff --git a/README.md b/README.md
index e428528..a80a0fa 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,2 @@
 # Prankbot
 -A script that visit youtube when ran hehe
 +The prank script is a Python program designed to visit YouTube in a way that mimics normal human behaviour, without making it obvious
 library to open a web browser, navigate to YouTube, scroll through the page, and optionally perform a search. By randomising the pauses
```

We found his email.

Next step is contact him with this email:



Finally, he reply to me with the flag.



Crypto – Hidden Prime

CHALLENGE

1 SOLVES



HIDDEN PRIME

★ 500

Challenge author: pikaroot

 dist.zip

Flag

Submit

```
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 n_2). 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: n_1 :

```
169826694190963923927267278920506304774712936009771387964060996166650366233
809890947300475518309943342695272769257877183849522549164411963837361339966
082070773067493543446928782957635581298495261109803286446607762230077270514
783555692694503905573681919274317442010927011119078878207456900633227358543
073819285432412696774748173293404058603479398230246605586112177667389233492
602044073041448316281411106607245065716279422649534567137352459449604013654
866853608881034777933788526234904892122909162245026425009231779843874824473
375035079931982808264699404984183250925406930183832251769332945894238185570
03478593666138697
```

c_1 :

```
412189728506967253328946206283776200483696356707426390359543305848480260234
942002545828672513025439369846592510636235279053179239041732077920533642590
735458091698001397213413222502715150810495909795820273823402735739897031751
104126820024530782349626394393704951383048593795951230369558530849274950536
898809095992490753598630383911932206182744171833305640566771512875402220195
234172020723996516582948633536169915172658915806002084930484973775034705002
327045091849207068766021999819842107135159741503674844171901861870857078437
423571222127646473070907094480661485125714271451574314789741859963756058588
2481622156452959
```

c_2 :

```
643782164278704828516303782284236318881990108461153326767749180428925087367
401155299799035187352494858869198822405158822531038272779302217480316224728
705589616570032018893964696043591165327338686618152076700540706714238456629
528528440965871993656855902718403069225931746243414798063853096689988748826
727871743579473382588394694457367977507870040471973444964228150393929016946
877281079308935984091851373289924135817572259304240757142316272869487944976
801243442657488363061520077078958732050939152313758395737967052963302308945
087235146838021919613913050221421547406644304722051762074443135150395085166
5477608448311216
```

hint:

```
353015607105234557942386719948921968517612921135279710920134842707780251056
055531103708310272177206965256187983229215588986036431249388608509798562422
0508184371060437167735952689307599
```

There is:

2 ciphertexts: c_1 and c_2




2 moduli: n_1 and n_2

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



Run




Output

```
1 from math import gcd
2
3 # Given constants
4 n1 = 1698266941909639239272672789205063047747129360097713879640609961666503662
  33809890947300475518309943342695272769257877183849522549164411963837361339
  96608207077306749354344692878295763558129849526110980328644660776223007727
  05147835556926945039055736819192743174420109270111190788782074569006332273
  58543073819285432412696774748173293404058603479398230246605586112177667389
  23349260204407304144831628141110660724506571627942264953456713735245944960
  40136548668536088810347779337885262349048921229091622450264250092317798438
  74824473375035079931982808264699404984183250925406930183832251769332945894
  23818557003478593666138697
5
6 c1 = 4121897285069672533289462062837762004836963567074263903595433058484802602
  34942002545828672513025439369846592510636235279053179239041732077920533642
  59073545809169800139721341322250271515081049590979582027382340273573989703
  17511041268200245307823496263943937049513830485937959512303695585308492749
  50536898809095992490753598630383911932206182744171833305640566771512875402
  22019523417202072399651658294863353616991517265891580600208493048497377503
  47050023270450918492070687660219998198421071351597415036748441719018618708
  57078437423571222127646473070907094480661485125714271451574314789741859963
  7560585882481622156452959
7
8 c2 = 6437821642787048285163037822842363188819901084611533267677491804289250873
  67401155299799035187352494858869198822405158822531038272779302217480316224
9
10 hint = 35301560710523455794238671994892196851761292113527971092013484270778025
  10560555311037083102721772069652561879832292155889860364312493886085097985
  62422050818437106043716773595268930759967262651453444254993228571959214130
  264090031153467025525843052563321209040817227102192868986669852039800541
```

```
gctf{sh4r3d_prim3s_ar3_cr4zy!!!}

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

main.py



Run

Output

```
5
6 c1 = 4121897285069672533289462062837762004836963567074263903595433058484802602
  34942002545828672513025439369846592510636235279053179239041732077920533642
  59073545809169800139721341322250271515081049590979582027382340273573989703
  17511041268200245307823496263943937049513830485937959512303695585308492749
  50536898809095992490753598630383911932206182744171833305640566771512875402
  22019523417202072399651658294863353616991517265891580600208493048497377503
  47050023270450918492070687660219998198421071351597415036748441719018618708
  57078437423571222127646473070907094480661485125714271451574314789741859963
  7560585882481622156452959
7
8 c2 = 6437821642787048285163037822842363188819901084611533267677491804289250873
  67401155299799035187352494858869198822405158822531038272779302217480316224
  72870558961657003201889396469604359116532733868661815207670054070671423845
  66295285284409658719936568559027184030692259317462434147980638530966899887
  48826727871743579473382588394694457367977507870040471973444964228150393929
  01694687728107930893598409185137328992413581757225930424075714231627286948
  79449768012434426574883630615200707789587320509391523137583957379670529633
  02208945087235146838021919613913050221421547406644304722051762074443135150
  3950851665477608448311216
9
10 hint = 35301560710523455794238671994892196851761292113527971092013484270778025
  10560555311037083102721772069652561879832292155889860364312493886085097985
  62422050818437106043716773595268930759967262651453444254993228571959214130
  264090031153467025525843052563321209040817227102192868986669852039800541
```

```
gctf{sh4r3d_prim3s_ar3_cr4zy!!!}

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

main.py

Share

Run

Output

```
9
10 hint = 35301560710523455794238671994892196851761292113527971092013484270778025
    10560555311037083102721772069652561879832292155889860364312493886085097985
    62422050818437106043716773595268930759967262651453444254993228571959214130
    264090031153467025525843052563321209040817227102192868986669852039800541
    42086527608953489442374266871037265656133266288384632862230562937377817623
    48692165311936675400876656603132061204963736368734246808096988732236926680
    25586257096871543752086660200691342766826024186847510790006449674795802360
    61946900552590955621115053260633715428080838629248392832413482185618881612
    94996285254457655139379079118821394059529712628137504540364788607048628547
    64846371687427901676018611325009792360072676788761698564467836032725211236
    9594474332604164577392228228377247268534657806928731011465532635507083237
    3471840804207640483749541597385759366927740974

11
12 # Step 1: Calculate the common factor x
13 E = hint // (n1 % hint)
14 n2 = hint - (n1 * E)
15 x = gcd(n1, n2)
16
17 # Step 2: Retrieve the prime factors p and q
18 p = n1 // x
19 q = n2 // x
20
21 # Step 3: Calculate φ(n) = (p-1)(q-1)
22 phi_n1 = (p - 1) * (x - 1) # Corrected to use x instead of q
```

gctf{sh4r3d_prim3s_ar3_cr4zy!!!}

=== Code Execution Successful ===

main.py

Share

Run

Output

```
..
12 # Step 1: Calculate the common factor x
13 E = hint // (n1 % hint)
14 n2 = hint - (n1 * E)
15 x = gcd(n1, n2)
16
17 # Step 2: Retrieve the prime factors p and q
18 p = n1 // x
19 q = n2 // x
20
21 # Step 3: Calculate φ(n) = (p-1)(q-1)
22 phi_n1 = (p - 1) * (x - 1) # Corrected to use x instead of q
23
24 # Step 4: Compute the private key d using the modular inverse
25 e = 0x10001
26 d1 = pow(e, -1, phi_n1)
27
28 # Step 5: Decrypt the ciphertexts
29 flag1 = pow(c1, d1, n1)
30
31 # We need to calculate φ(n2) and d2 as well
32 phi_n2 = (q - 1) * (x - 1) # Calculate φ(n2)
33 d2 = pow(e, -1, phi_n2) # Compute private key for n2
34
35 flag2 = pow(c2, d2, n2) # Decrypt c2 with d2
36
```

gctf{sh4r3d_prim3s_ar3_cr4zy!!!}

=== Code Execution Successful ===

main.py

Share

Run

Output

```
32 phi_n2 = (q - 1) * (x - 1) # Calculate φ(n2)
33 d2 = pow(e, -1, phi_n2) # Compute private key for n2
34
35 flag2 = pow(c2, d2, n2) # Decrypt c2 with d2
36
37 # Convert long integers back to bytes
38 def long_to_bytes(n):
39     """Convert a long integer to a byte string."""
40     if n == 0:
41         return b''
42     byte_arr = bytearray()
43     while n:
44         byte_arr.append(n & 0xff)
45         n >>= 8
46     return bytes(reversed(byte_arr))
47
48 flag1_bytes = long_to_bytes(flag1)
49 flag2_bytes = long_to_bytes(flag2)
50
51 # Combine both parts and attempt to decode
52 full_flag = flag1_bytes + flag2_bytes
53
54 # Handle potential decoding errors
55 try:
56     decoded_flag = full_flag.decode('utf-8')
```

gctf{sh4r3d_prim3s_ar3_cr4zy!!!}

=== Code Execution Successful ===

```
main.py  [Icons] [Share] [Run] Output
39  """Convert a long integer to a byte string."""
40  if n == 0:
41      return b''
42  byte_arr = bytearray()
43  while n:
44      byte_arr.append(n & 0xff)
45      n >>= 8
46  return bytes(reversed(byte_arr))
47
48 flag1_bytes = long_to_bytes(flag1)
49 flag2_bytes = long_to_bytes(flag2)
50
51 # Combine both parts and attempt to decode
52 full_flag = flag1_bytes + flag2_bytes
53
54 # Handle potential decoding errors
55 try:
56     decoded_flag = full_flag.decode('utf-8')
57 except UnicodeDecodeError:
58     decoded_flag = full_flag.decode('latin-1') # Try a different encoding
59
60 # Return the decrypted flag
61 print(decoded_flag)
62
```

gctf{sh4r3d_prim3s_ar3_cr4zy!!!}

=== Code Execution Successful ===

Code Explanation

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

$E = \text{hint} // (n1 \% \text{hint})$

$n2 = \text{hint} - (n1 * E)$

$x = \text{gcd}(n1, n2)$

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 xxx 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 $\phi(n)$

$\phi_{n1} = (p - 1) * (x - 1)$

$\phi_{n2} = (q - 1) * (x - 1)$

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

Step 4: Compute the Private Key

$d1 = \text{pow}(e, -1, \phi_{n1})$ # Private key for $n1$

$d2 = \text{pow}(e, -1, \phi_{n2})$ # Private key for $n2$

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

Step 5: Decrypt the Ciphertexts

$\text{flag1} = \text{pow}(c1, d1, n1)$

$\text{flag2} = \text{pow}(c2, d2, n2)$

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

$\text{full_flag} = \text{flag1_bytes} + \text{flag2_bytes}$

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

5 SOLVES

✕

STOP SLACKING OFF

★ 436

My boss loves to spread rumours and give bad jokes through Slack, I wonder if he have anything to say about me.

Note: Forensics files were given before the CTF. If you haven't downloaded them, here's the link: <https://shorturl.at/bHfhc>

ZIP Password:
2e83021cacb2aa2dc180901b31514d856830858f

Challenge author: warlocksmurf

Submit

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.

Searching for "joke":

The screenshot shows a Windows desktop with a Notepad++ editor open. The editor's title bar indicates the file is 'config.inc.ppt'. The JSON content is as follows:

```

{
  "messages": [
    {
      "type": "message",
      "text": "I'm still waiting for the secret code btw",
      "block_id": "9vaw",
      "channel": "C07LBGMD94L",
      "user": "U07LBGHSQKE",
      "rxn_key": "message-1725626871.191789-C07LBGMD94L",
      "subtype": "text",
      "meta_o": "oh I forgot to send the emails lol",
      "last_updated_ts": "10739.199999999953",
      "thread_ts": "1725626880.261809o",
      "hidden_reply_f": "reply_count",
      "latest_reply": "replies",
      "reply_users_count": "files",
      "attachments": [
        {
          "type": "rich_text",
          "block_id": "Awuq",
          "rich_text": {
            "block_id": "Awuq",
            "text": "oh I forgot to send the emails lol"
          }
        }
      ],
      "client_msg_id": "3205942c-0745-414c-ab2d-81b1a5117f55",
      "source_team_id": "T07LMPXLCN9"
    },
    {
      "type": "message",
      "text": "I'm still waiting for the secret code btw",
      "block_id": "9vaw",
      "channel": "C07LBGMD94L",
      "user": "U07LBGHSQKE",
      "rxn_key": "message-1725626880.261809-C07LBGMD94L",
      "subtype": "text",
      "meta_o": "oh I forgot to send the emails lol",
      "last_updated_ts": "10739.199999999953",
      "thread_ts": "1725626907.802259o",
      "hidden_reply_f": "reply_count",
      "latest_reply": "replies",
      "reply_users_count": "files",
      "attachments": [
        {
          "type": "rich_text",
          "block_id": "Bz9x",
          "rich_text": {
            "block_id": "Bz9x",
            "text": "you wanna hear another joke?"
          }
        }
      ],
      "client_msg_id": "562cd219-86dd-4360-9e00-b07db5418845",
      "source_team_id": "T07LMPXLCN9"
    }
  ]
}

```

A red box highlights the 'lackbot_feels0' message in the JSON array. A search bar is visible at the top of the editor, showing the search term 'joke'.

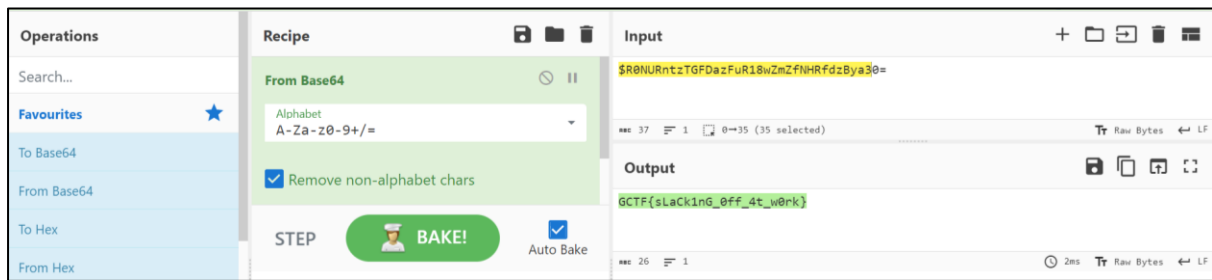
```

JAMUNELTPWAVE
reply_countI " replies_"
latest_reply_"
reply_users_"reply_users_count_"files_"
attachments_"blocksaI o"otype" rich_text"block_id"015zkW"elementsaI o"otype"rich_text_section"elementsaI o"otype"text"@"text"@"Fine, here is
the code"@"@"@"@"blocksProcessed@aI o"otype" rich_text" blockId"015zkW"elementsaI o"otype"rich_text_section"elementsaI
o"otype"text"@"text"@"Fine, here is the code"@"@"@"@"@"
client_msg_id"$2e4b6fb4-ac5b-480a-945f-bce961ef39a5"source_team_id"
T07LMPXLCN9"otype" message"@"ts"01725627840.667679" channel"
C07LBGMD94L"
no_displayF"User"
U07LBGHTV44"rxn_key"%message-1725627840.667679-C07LBGMD94L" subtype"@"text"@"Fine, here is the code"@"_meta_o"
lastUpdatedTs"010739.299999999814{"@"@"01725627841.855489o" thread_ts_"slackbot_fees10
_hidden_replyF"
reply_countI " replies_"
latest_reply_"
reply_users_"reply_users_count_"files_"
attachments_"blocksaI o"otype" rich_text"block_id"08kccF"elementsaI o"otype"rich_text_section"elementsaI
o"otype"text"@"text"$R0NURntzTGFdZuFur18wZmZmNHRfdzBya3o{"@"@"@"@"blocksProcessed@aI o"otype" rich_text" blockId"08kccF"elementsaI
o"otype"rich_text_section"elementsaI o"otype"text"@"text"$R0NURntzTGFdZuFur18wZmZmNHRfdzBya3o{"@"@"@"@"@"
client_msg_id"$e9917fb3-e64f-4188-922e-70bc03aee6d"source_team_id"
T07LMPXLCN9"otype" message"@"ts"01725627841.855489" channel"
C07LBGMD94L"
no_displayF"User"

```

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}**