Week4- Write-up

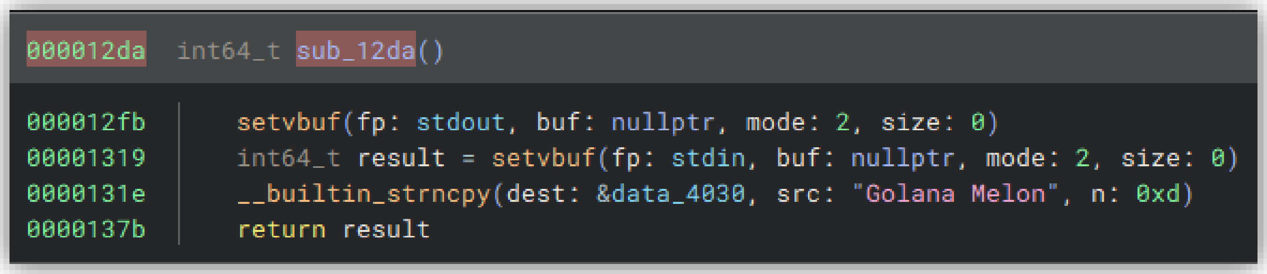
Vishnu Vardhan Ciripuram
N14912012
vc2499

Challenge: Stripped

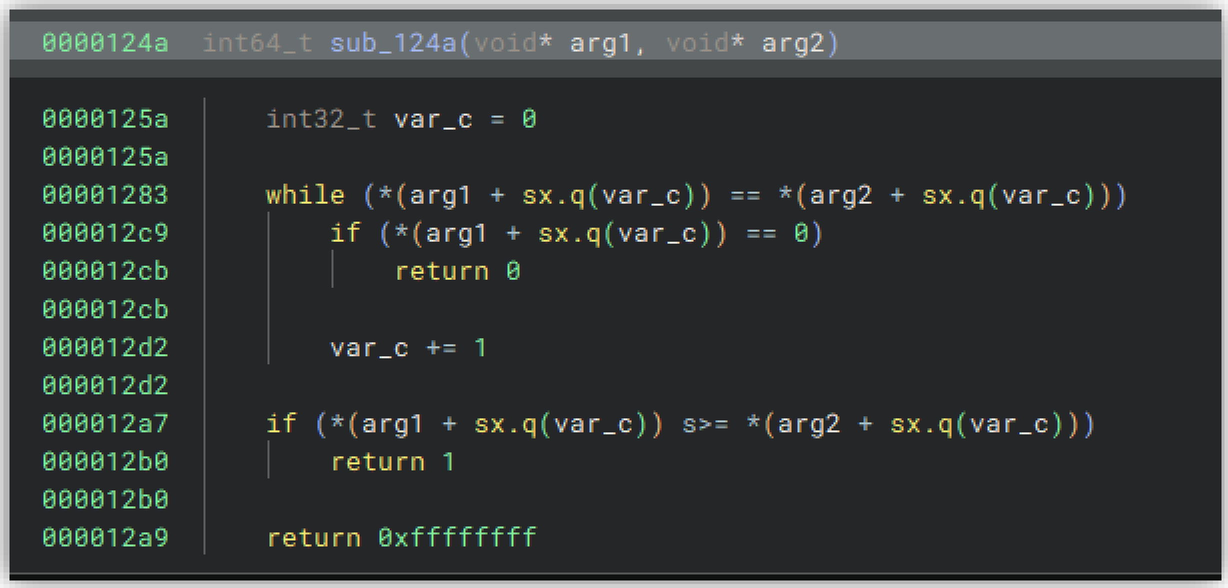
The challenge required navigating through a binary protocol by analyzing the program's logic using Binary Ninja and understanding syscall-based function calls.

Solution Steps:

- I began by analyzing the **main** function (**sub_137c**) and noticed various function calls:
    - **sub_12da**: This function initialized the program and set up strings like "Golana Melon".
    - **sub_124a**: This function compared the input strings just like **strcmp.**

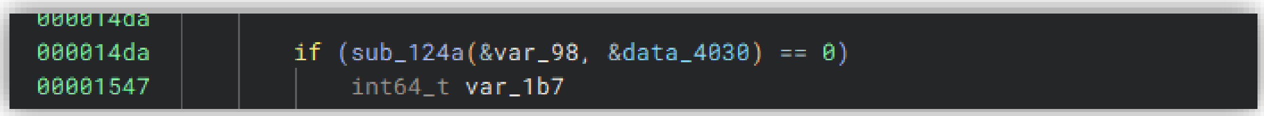


```
000012da  int64_t sub_12da()

000012fb      setvbuf(fp: stdout, buf: nullptr, mode: 2, size: 0)
00001319      int64_t result = setvbuf(fp: stdin, buf: nullptr, mode: 2, size: 0)
0000131e      __builtin_strncpy(dest: &data_4030, src: "Golana Melon", n: 0xd)
0000137b      return result
```



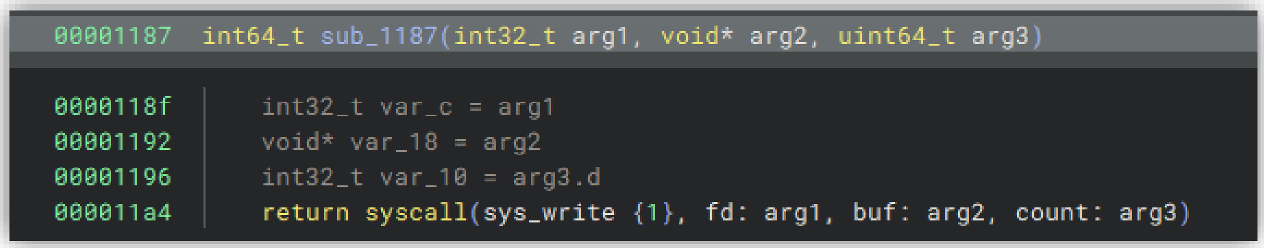
```
0000124a  int64_t sub_124a(void* arg1, void* arg2)

0000125a      int32_t var_c = 0
0000125a
00001283      while (*(arg1 + sx.q(var_c)) == *(arg2 + sx.q(var_c)))
000012c9          if (*(arg1 + sx.q(var_c)) == 0)
000012cb              return 0
000012cb
000012d2          var_c += 1
000012d2
000012a7      if (*(arg1 + sx.q(var_c)) s>= *(arg2 + sx.q(var_c)))
000012b0          return 1
000012b0
000012a9      return 0xffffffff
```

- I saw that the first comparison using **sub_124a** checked the input against "Golana Melon" stored at **data_4030**.

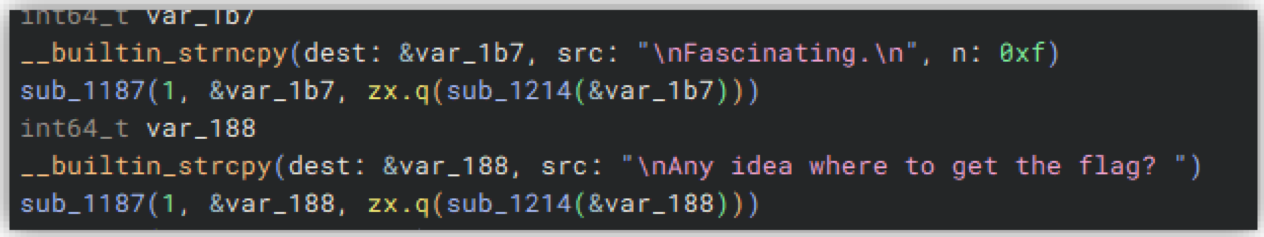


```
000014da
000014da        if (sub_124a(&var_98, &data_4030) == 0)
00001547            int64_t var_1b7
```

- After the input matched "Golana Melon", the program called **sub_1187** (a wrapper for **sys_write**) to print "Fascinating." and prompted me with "Any idea where to get the flag?"

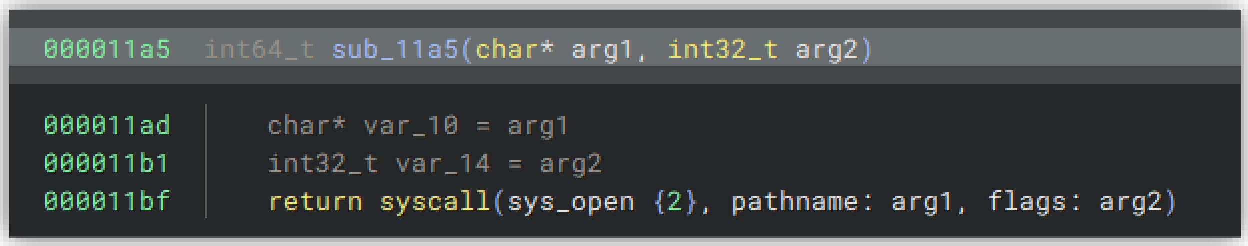


```
00001187  int64_t sub_1187(int32_t arg1, void* arg2, uint64_t arg3)

0000118f    int32_t var_c = arg1
00001192    void* var_18 = arg2
00001196    int32_t var_10 = arg3.d
000011a4    return syscall(sys_write {1}, fd: arg1, buf: arg2, count: arg3)
```



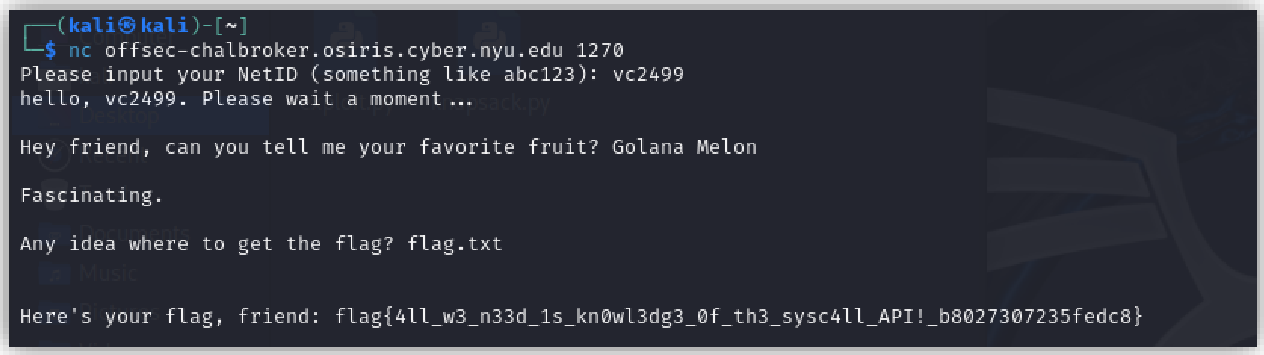
```
int64_t var_1b7
__builtin_strncpy(dest: &var_1b7, src: "\nFascinating.\n", n: 0xf)
sub_1187(1, &var_1b7, zx.q(sub_1214(&var_1b7)))
int64_t var_188
__builtin_strcpy(dest: &var_188, src: "\nAny idea where to get the flag? ")
sub_1187(1, &var_188, zx.q(sub_1214(&var_188)))
```

- I noticed that the next input was passed to **sub_11a5 (sys_open),** which attempted to open a file specified by the user.



```
000011a5  int64_t sub_11a5(char* arg1, int32_t arg2)

000011ad    char* var_10 = arg1
000011b1    int32_t var_14 = arg2
000011bf    return syscall(sys_open {2}, pathname: arg1, flags: arg2)
```

- The first argument to **sys_open** is typically the pathname. This indicated that the expected answer was the filename **flag.txt**.
- I submitted this to the server and received the flag: **flag{4ll_w3_n33d_1s_kn0wl3dg3_0f_th3_sysc4ll_API!_b8027307235fedc8}**



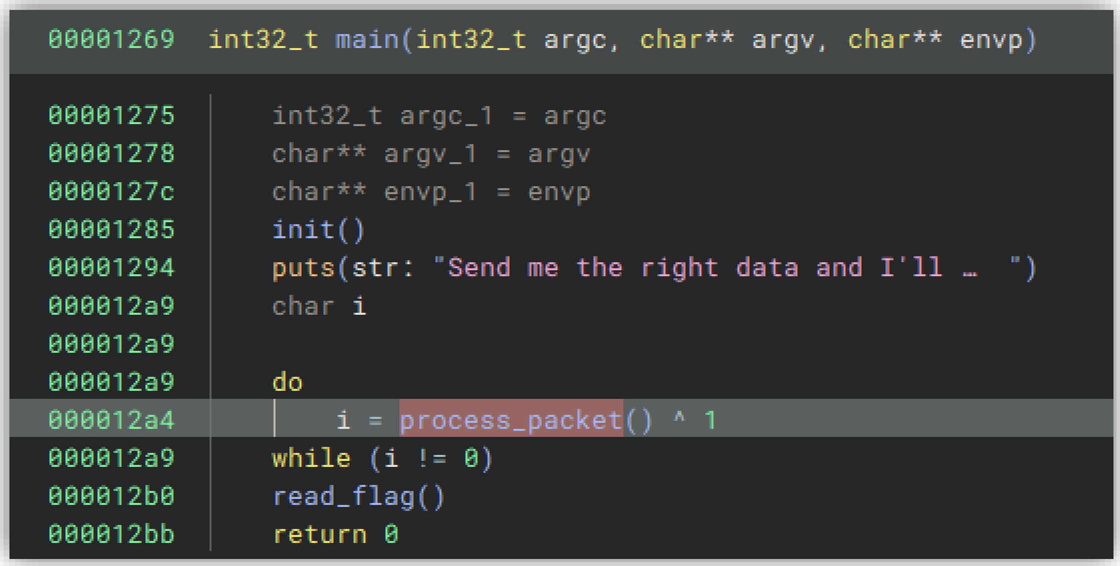
```
┌──(kali㉿kali)-[~]
└─$ nc offsec-chalbroker.osiris.cyber.nyu.edu 1270
Please input your NetID (something like abc123): vc2499
hello, vc2499. Please wait a moment...

Hey friend, can you tell me your favorite fruit? Golana Melon

Fascinating.

Any idea where to get the flag? flag.txt


Here's your flag, friend: flag{4ll_w3_n33d_1s_kn0wl3dg3_0f_th3_sysc4ll_API!_b8027307235fedc8}
```

Challenge: Rudimentary Data Protocol

The challenge required understanding a custom binary protocol by analyzing the binary and determining the specific packet structure and commands required to trigger the correct responses.

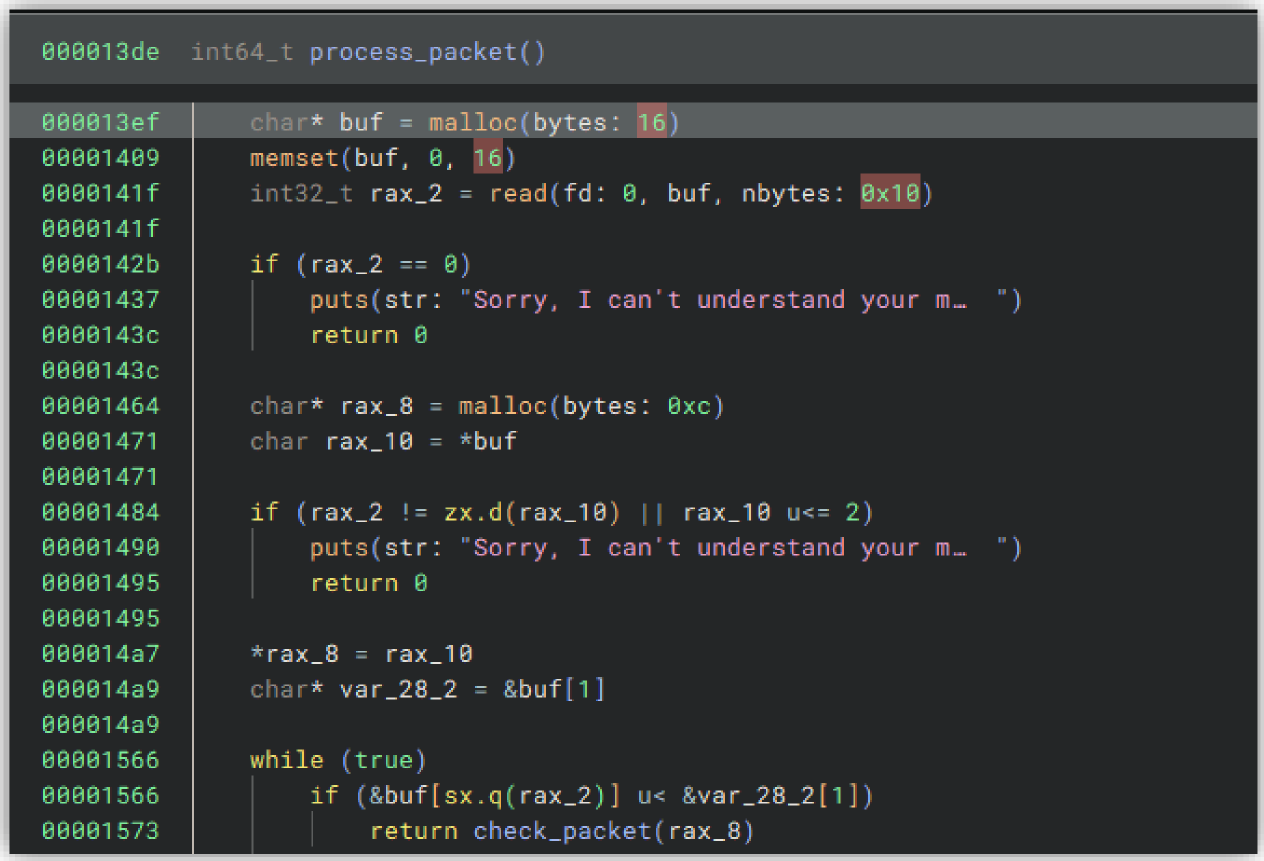
Solution Steps:

- I started by analyzing the main function (sub_1269) using Binary Ninja. The function had a loop that repeatedly called process_packet() until a condition was met.



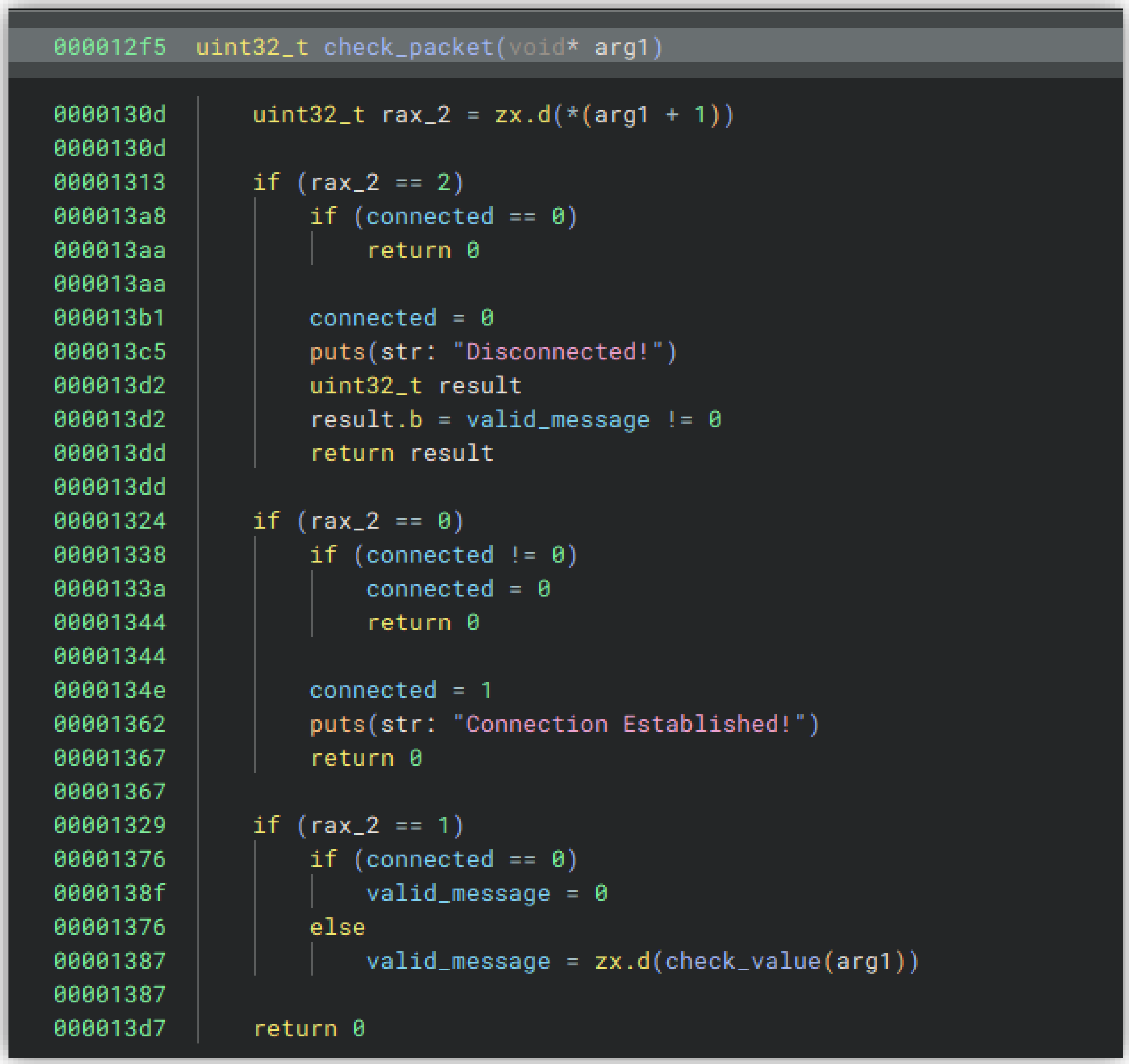
```
00001269  int32_t main(int32_t argc, char** argv, char** envp)

00001275      int32_t argc_1 = argc
00001278      char** argv_1 = argv
0000127c      char** envp_1 = envp
00001285      init()
00001294      puts(str: "Send me the right data and I'll …  ")
000012a9      char i
000012a9
000012a9      do
000012a4          i = process_packet() ^ 1
000012a9      while (i != 0)
000012b0      read_flag()
000012bb      return 0
```

- I then examined the **process_packet** function (sub_13de). The function read up to **16 bytes** into a buffer. The first byte (buf[0]) was used as the **length** of the packet, and the second byte (buf[1]) determined the **command** type. This suggested that each packet should have a minimum length of 3 bytes.



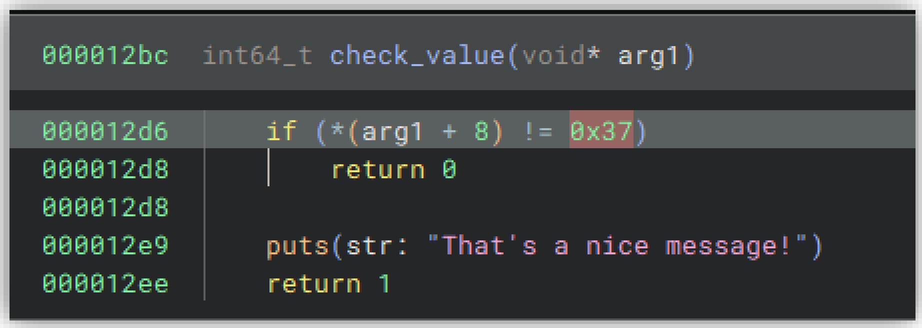
```
000013de  int64_t process_packet()

000013ef      char* buf = malloc(bytes: 16)
00001409      memset(buf, 0, 16)
0000141f      int32_t rax_2 = read(fd: 0, buf, nbytes: 0x10)
0000141f
0000142b      if (rax_2 == 0)
00001437          puts(str: "Sorry, I can't understand your m…  ")
0000143c          return 0
0000143c
00001464      char* rax_8 = malloc(bytes: 0xc)
00001471      char rax_10 = *buf
00001471
00001484      if (rax_2 != zx.d(rax_10) || rax_10 u<= 2)
00001490          puts(str: "Sorry, I can't understand your m…  ")
00001495          return 0
00001495
000014a7      *rax_8 = rax_10
000014a9      char* var_28_2 = &buf[1]
000014a9
00001566      while (true)
00001566          if (&buf[sx.q(rax_2)] u< &var_28_2[1])
00001573              return check_packet(rax_8)
```

- If the packet didn't meet the expected format, it printed **"Sorry, I can't understand your message."** and returned 0.
- I then moved on to analyze **check_packet** (sub_13f5), which handled different commands based on the **second byte** of the packet (buf[1]):
    - **0x00** (Connect): Sets connected = 1 and prints **"Connection Established!"**.
    - **0x01** (Send Value): Calls check_value to verify if the value in the packet is 0x37.
    - **0x02** (Disconnect): Sets connected = 0 and checks if valid_message is 1 before printing the flag.



```
000012f5  uint32_t check_packet(void* arg1)

0000130d      uint32_t rax_2 = zx.d(*(arg1 + 1))
0000130d
00001313      if (rax_2 == 2)
000013a8          if (connected == 0)
000013aa              return 0
000013aa
000013b1          connected = 0
000013c5          puts(str: "Disconnected!")
000013d2          uint32_t result
000013d2          result.b = valid_message != 0
000013dd          return result
000013dd
00001324      if (rax_2 == 0)
00001338          if (connected != 0)
0000133a              connected = 0
00001344              return 0
00001344
0000134e          connected = 1
00001362          puts(str: "Connection Established!")
00001367          return 0
00001367
00001329      if (rax_2 == 1)
00001376          if (connected == 0)
0000138f              valid_message = 0
00001376          else
00001387              valid_message = zx.d(check_value(arg1))
00001387
000013d7      return 0
```

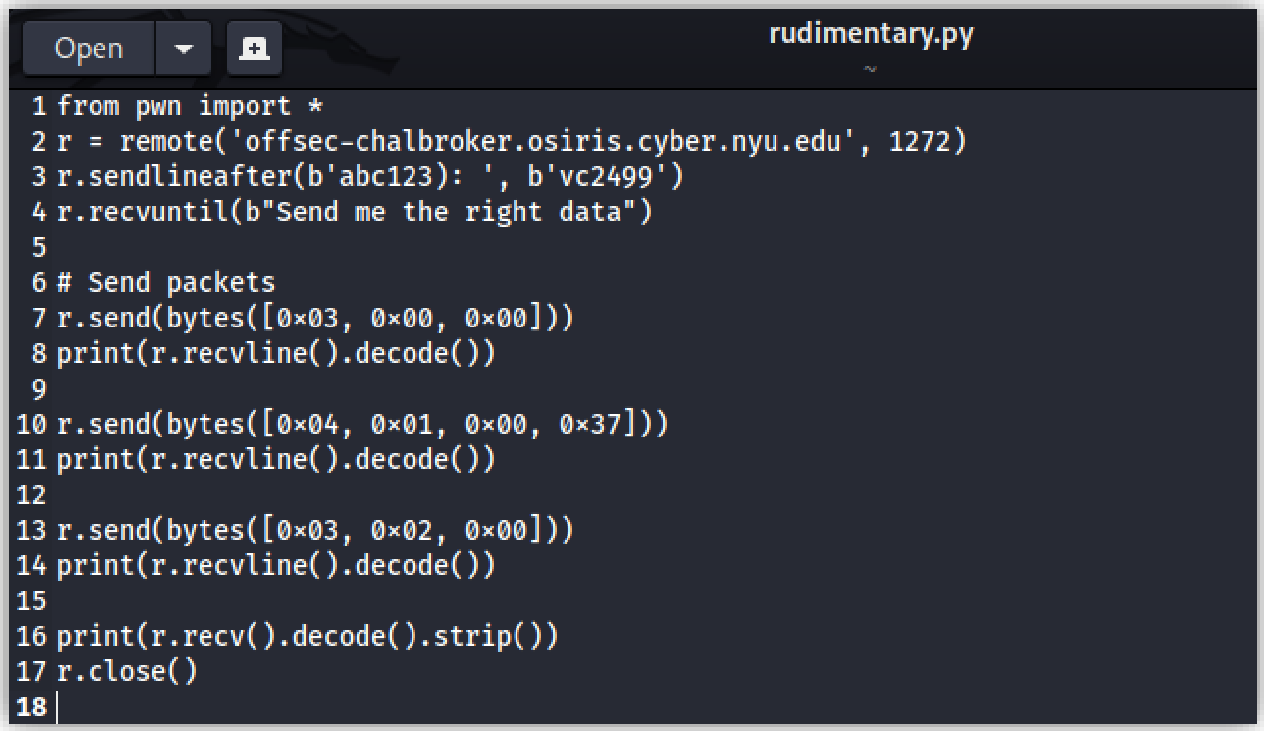
- I then focused on the **check_value** function (**sub_12bc**). It checks *(arg1 + 8) == 0x37. Here, arg1 points to the start of the packet (buf), so ***(arg1 + 8)** refers to the 4th byte of buf (rax_8[8]) equals 0x37.
- For the **Send Value** packet to pass the check_value check, we need the fourth byte (buf[3]) to be 0x37, for it to satisfy the condition and sets valid_message to 1.



```
000012bc  int64_t check_value(void* arg1)

000012d6      if (*(arg1 + 8) != 0x37)
000012d8          return 0
000012d8
000012e9      puts(str: "That's a nice message!")
000012ee      return 1
```

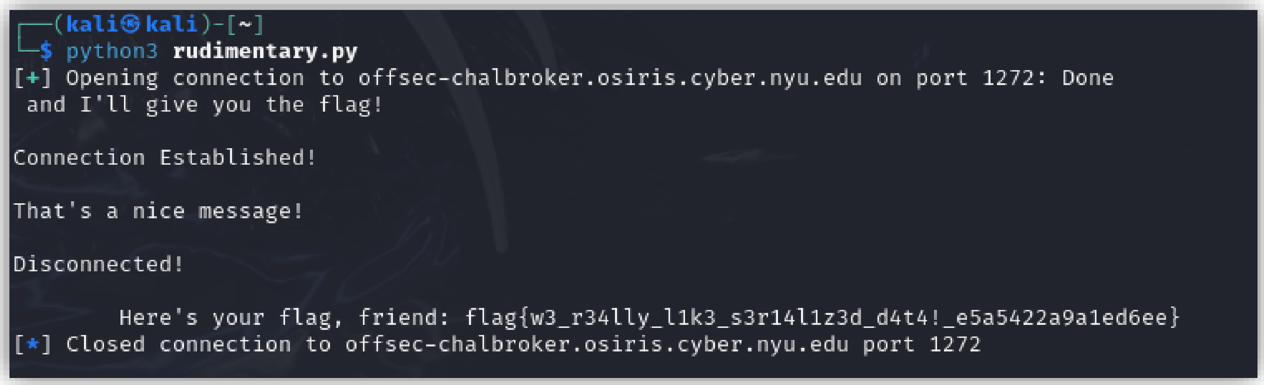
- Based on the analysis, I constructed the following packets:
- **Connect Packet**:
    - **Length**: 0x03 (3 bytes)
    - **Command**: 0x00 (Connect)
    - **Data**: Unused (0x00)
    - **Bytes**: [0x03, 0x00, 0x00]
- **Send Value Packet**:
    - **Length**: 0x04 (4 bytes)
    - **Command**: 0x01 (Send Value)
    - **Data**: Unused (0x00)
    - **Data**: 0x37 (the value expected by check_value)
    - **Bytes**: [0x04, 0x01, 0x00, 0x37]
- **Disconnect Packet**:
    - **Length**: 0x03 (3 bytes)
    - **Command**: 0x02 (Disconnect)
    - **Data**: Unused (0x00)
    - **Bytes**: [0x03, 0x02, 0x00]

- I wrote a python script to automate this process



```python
from pwn import *
r = remote('offsec-chalbroker.osiris.cyber.nyu.edu', 1272)
r.sendlineafter(b'abc123): ', b'vc2499')
r.recvuntil(b"Send me the right data")

# Send packets
r.send(bytes([0x03, 0x00, 0x00]))
print(r.recvline().decode())

r.send(bytes([0x04, 0x01, 0x00, 0x37]))
print(r.recvline().decode())

r.send(bytes([0x03, 0x02, 0x00]))
print(r.recvline().decode())

print(r.recv().decode().strip())
r.close()

```

- I ran the script and it game me the flag:
  flag{w3_r34lly_l1k3_s3r14l1z3d_d4t4!_e5a5422a9a1ed6ee}



```
┌──(kali㉿kali)-[~]
└─$ python3 rudimentary.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1272: Done
 and I'll give you the flag!

Connection Established!

That's a nice message!

Disconnected!

        Here's your flag, friend: flag{w3_r34lly_l1k3_s3r14l1z3d_d4t4!_e5a5422a9a1ed6ee}
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1272
```

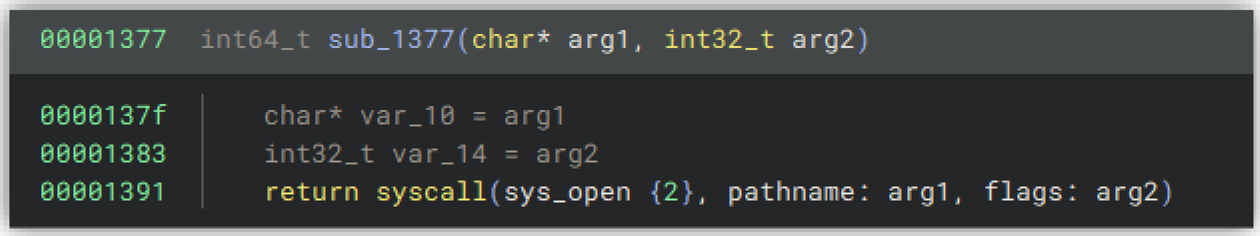
Challenge: Hand Rolled Cryptex

This challenge required navigating through a multi-phase binary prompt, providing specific inputs to bypass each check and reveal the final flag.

Solution Steps:

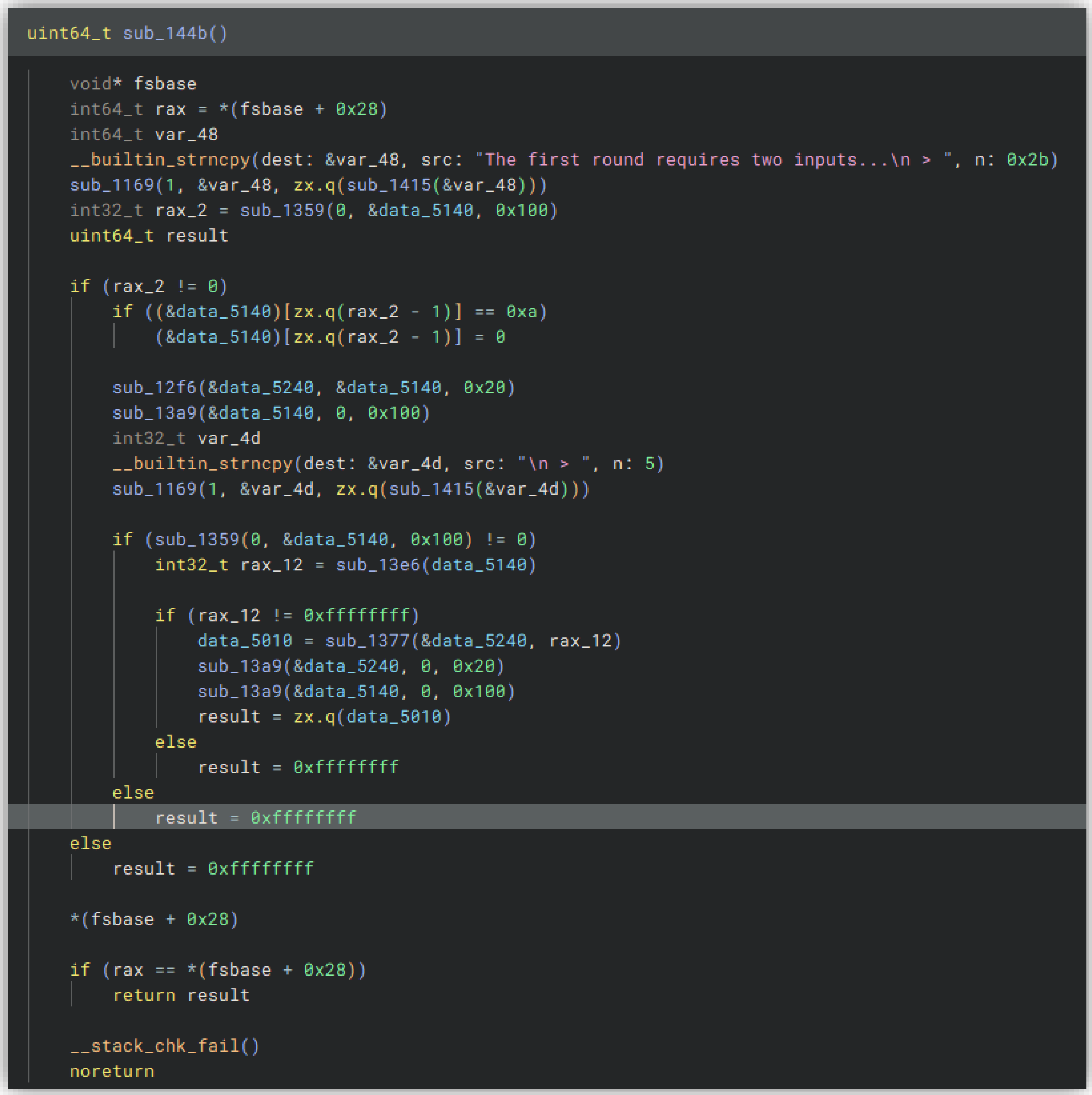
Q1: "The first round requires two inputs..."

- I started by loading the binary into **Binary Ninja** to analyze its internal logic.
- Inside the function sub_144b, I found that it prompts the user for **two inputs**:
  - First value stored in the **data_5140** buffer.
  - second value stored in the **data_5240** buffer.
- The function sub_1359 reads input into data_5140 and checks if the input ends with a newline character (\n), removing it if present.
- The function sub_1377 attempts to **open a file** using the input stored in **data_5240** with flags specified by the **second input**. The result of this open operation is stored in data_5010.



```
00001377  int64_t sub_1377(char* arg1, int32_t arg2)

0000137f      char* var_10 = arg1
00001383      int32_t var_14 = arg2
00001391      return syscall(sys_open {2}, pathname: arg1, flags: arg2)
```

- This tells us that the first input is being used as the file path that the program is trying to open. The first argument to **sys_open** is typically the pathname. This indicated that the expected answer was the filename **flag.txt**.
- After reading the second input, the program calls sub_13e6, it converts a character to its numeric value. The numeric value (rax_12) is passed as the **second argument** to sub_1377
- To **read** the contents of flag.txt, the **file access mode should be 0.**
  - **File Path**: "flag.txt"
  - **Flag Digit**: 0 (indicates read permissions).

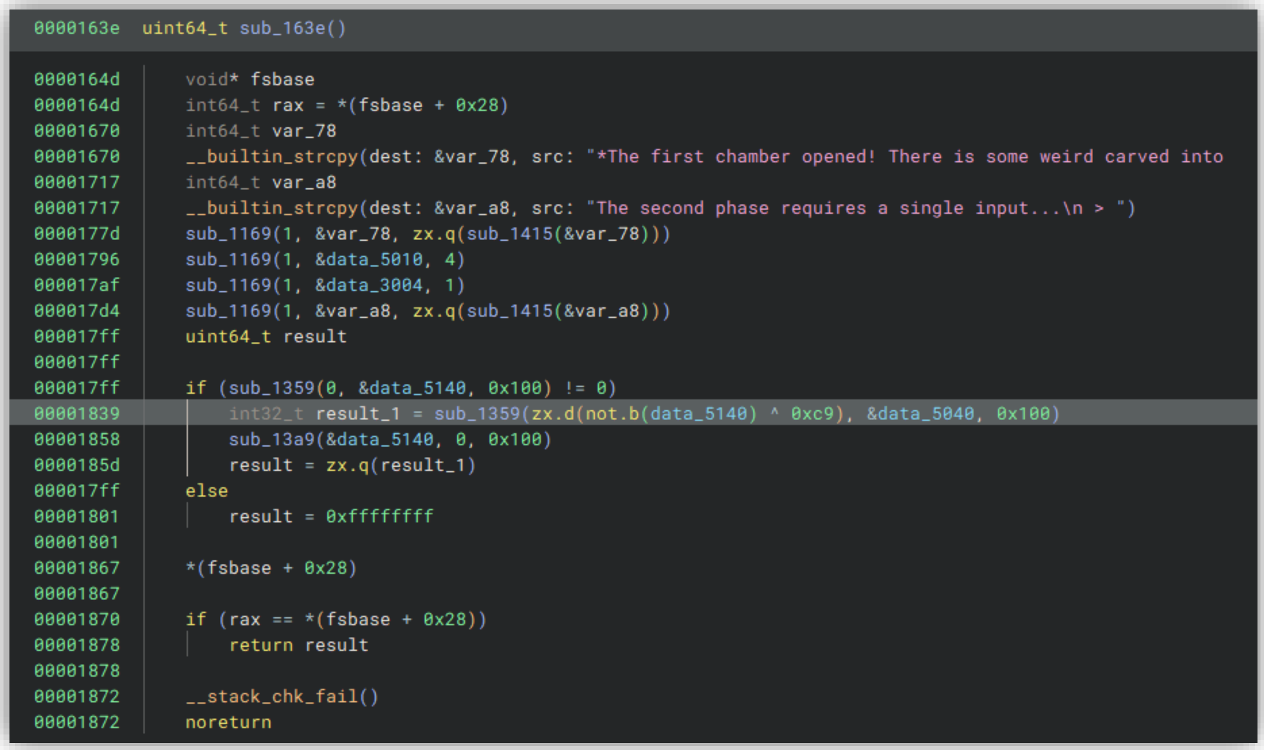


```
uint64_t sub_144b()

    void* fsbase
    int64_t rax = *(fsbase + 0x28)
    int64_t var_48
    __builtin_strncpy(dest: &var_48, src: "The first round requires two inputs...\n > ", n: 0x2b)
    sub_1169(1, &var_48, zx.q(sub_1415(&var_48)))
    int32_t rax_2 = sub_1359(0, &data_5140, 0x100)
    uint64_t result

    if (rax_2 != 0)
        if ((&data_5140)[zx.q(rax_2 - 1)] == 0xa)
            (&data_5140)[zx.q(rax_2 - 1)] = 0

        sub_12f6(&data_5240, &data_5140, 0x20)
        sub_13a9(&data_5140, 0, 0x100)
        int32_t var_4d
        __builtin_strncpy(dest: &var_4d, src: "\n > ", n: 5)
        sub_1169(1, &var_4d, zx.q(sub_1415(&var_4d)))

        if (sub_1359(0, &data_5140, 0x100) != 0)
            int32_t rax_12 = sub_13e6(data_5140)

            if (rax_12 != 0xffffffff)
                data_5010 = sub_1377(&data_5240, rax_12)
                sub_13a9(&data_5240, 0, 0x20)
                sub_13a9(&data_5140, 0, 0x100)
                result = zx.q(data_5010)
            else
                result = 0xffffffff
        else
            result = 0xffffffff
    else
        result = 0xffffffff

    *(fsbase + 0x28)

    if (rax == *(fsbase + 0x28))
        return result

    __stack_chk_fail()
    noreturn
```

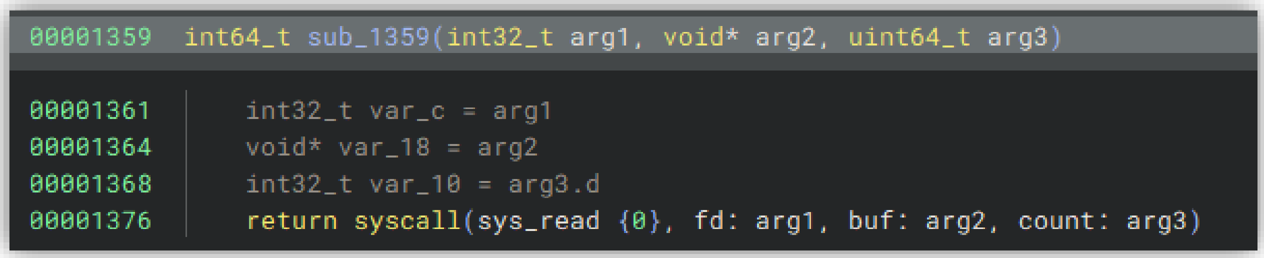
Q 2: "The second phase requires a single input..."

- The binary moved to the next phase in sub_163e, asking for a **single input**.
- I observed that the binary performs a validation check on the input using **sub_13e6**:
- The input byte is passed to **function sub_13e6**, where the binary checks if the character falls within the **ASCII range for digits** (0x30 to 0x39). If it does, it **subtracts 0x30** to get the **integer value**.If valid, it converts the input to its **integer equivalent** by subtracting 0x30 (ASCII '0').

- ASCII 0x30 ('0') becomes the integer value 0.
- The program then **XORs** this integer value with the constant 0xC9 and maps it to a **file descriptor** (fd):
- **fd = input_byte ^ 0xc9**
- fd takes the value 6 to access the flag **6 ^ 0xc9 = 0xcf** and when we mask it to be in range and we get **0**.
- **'0'** (ASCII 0x30), maps to the correct file descriptor 6 and allows the program to read the file successfully.
- **Input: '0'**



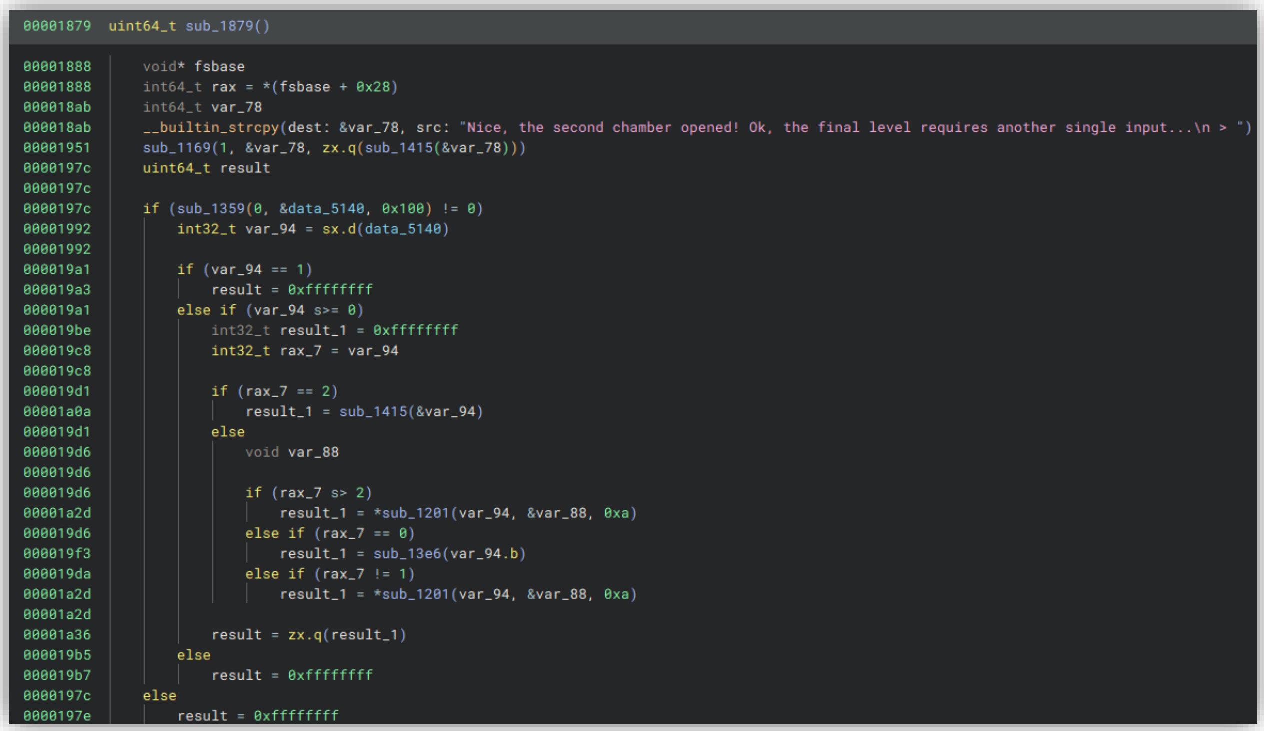
```
0000163e  uint64_t sub_163e()

0000164d      void* fsbase
0000164d      int64_t rax = *(fsbase + 0x28)
00001670      int64_t var_78
00001670      __builtin_strcpy(dest: &var_78, src: "*The first chamber opened! There is some weird carved into
00001717      int64_t var_a8
00001717      __builtin_strcpy(dest: &var_a8, src: "The second phase requires a single input...\n > ")
0000177d      sub_1169(1, &var_78, zx.q(sub_1415(&var_78)))
00001796      sub_1169(1, &data_5010, 4)
000017af      sub_1169(1, &data_3004, 1)
000017d4      sub_1169(1, &var_a8, zx.q(sub_1415(&var_a8)))
000017ff      uint64_t result
000017ff
000017ff      if (sub_1359(0, &data_5140, 0x100) != 0)
00001839          int32_t result_1 = sub_1359(zx.d(not.b(data_5140) ^ 0xc9), &data_5040, 0x100)
00001858          sub_13a9(&data_5140, 0, 0x100)
0000185d          result = zx.q(result_1)
000017ff      else
00001801          result = 0xffffffff
00001801
00001867      *(fsbase + 0x28)
00001867
00001870      if (rax == *(fsbase + 0x28))
00001878          return result
00001878
00001872      __stack_chk_fail()
00001872      noreturn
```



```
00001359  int64_t sub_1359(int32_t arg1, void* arg2, uint64_t arg3)

00001361      int32_t var_c = arg1
00001364      void* var_18 = arg2
00001368      int32_t var_10 = arg3.d
00001376      return syscall(sys_read {0}, fd: arg1, buf: arg2, count: arg3)
```

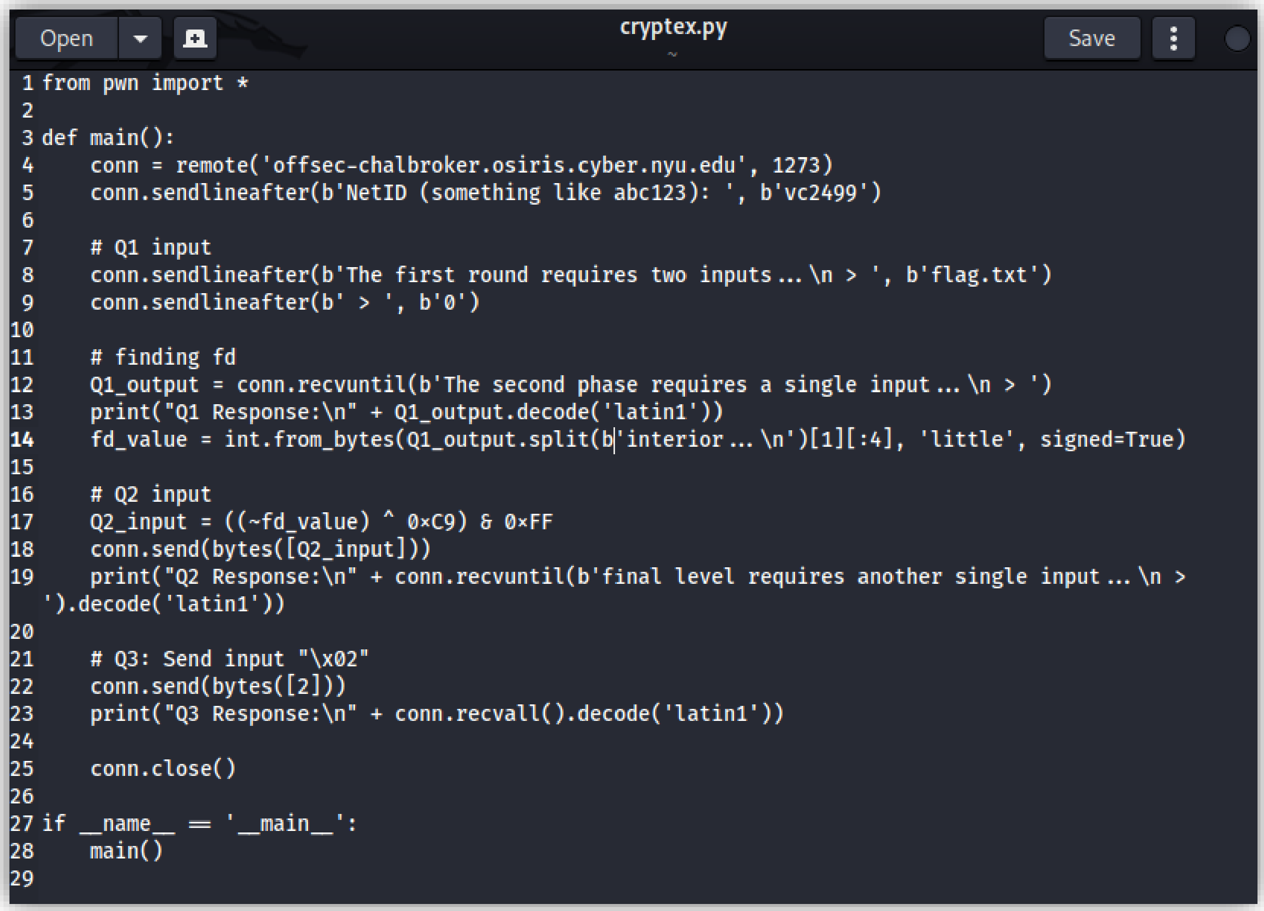
Q3: "The final level requires another single input..."

- The function **sub_1879** prompted for a **single-byte input**. Here, the input (var_94) was checked against specific conditions:
    - If var_94 == 1, the program triggers an error, making 1 an invalid input.
    - If var_94 == 2, the program calls sub_1415 and proceeds to access the correct file or reveal the flag
- Other values (var_94 > 2) invoke sub_1201, which performs additional operations, making these values unsuitable for obtaining the flag.
- To pass the check for var_94 == 2, I need to send the **raw byte b'\x02'** as the ASCII '2' (0x32) would be 50 and not satisfy this condition.
- **Input: b'\x02'**



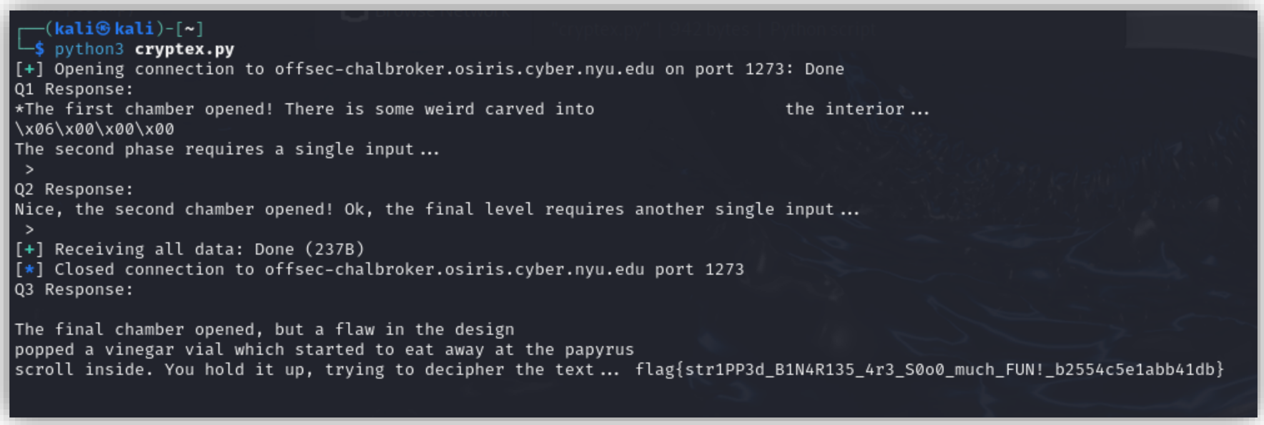
```
00001879  uint64_t sub_1879()

00001888      void* fsbase
00001888      int64_t rax = *(fsbase + 0x28)
000018ab      int64_t var_78
000018ab      __builtin_strcpy(dest: &var_78, src: "Nice, the second chamber opened! Ok, the final level requires another single input...\n > ")
00001951      sub_1169(1, &var_78, zx.q(sub_1415(&var_78)))
0000197c      uint64_t result
0000197c
0000197c      if (sub_1359(0, &data_5140, 0x100) != 0)
00001992          int32_t var_94 = sx.d(data_5140)
00001992
000019a1          if (var_94 == 1)
000019a3              result = 0xffffffff
000019a1          else if (var_94 s>= 0)
000019be              int32_t result_1 = 0xffffffff
000019c8              int32_t rax_7 = var_94
000019c8
000019d1              if (rax_7 == 2)
00001a0a                  result_1 = sub_1415(&var_94)
000019d1              else
000019d6                  void var_88
000019d6
000019d6                  if (rax_7 s> 2)
00001a2d                      result_1 = *sub_1201(var_94, &var_88, 0xa)
000019d6                  else if (rax_7 == 0)
000019f3                      result_1 = sub_13e6(var_94.b)
000019da                  else if (rax_7 != 1)
00001a2d                      result_1 = *sub_1201(var_94, &var_88, 0xa)
00001a2d
00001a36              result = zx.q(result_1)
000019b5          else
000019b7              result = 0xffffffff
0000197c      else
0000197e          result = 0xffffffff
```

- I wrote a python script to automate this process



```python
from pwn import *

def main():
    conn = remote('offsec-chalbroker.osiris.cyber.nyu.edu', 1273)
    conn.sendlineafter(b'NetID (something like abc123): ', b'vc2499')

    # Q1 input
    conn.sendlineafter(b'The first round requires two inputs...\n > ', b'flag.txt')
    conn.sendlineafter(b' > ', b'0')

    # finding fd
    Q1_output = conn.recvuntil(b'The second phase requires a single input...\n > ')
    print("Q1 Response:\n" + Q1_output.decode('latin1'))
    fd_value = int.from_bytes(Q1_output.split(b'interior...\n')[1][:4], 'little', signed=True)

    # Q2 input
    Q2_input = ((~fd_value) ^ 0xC9) & 0xFF
    conn.send(bytes([Q2_input]))
    print("Q2 Response:\n" + conn.recvuntil(b'final level requires another single input...\n > ').decode('latin1'))

    # Q3: Send input "\x02"
    conn.send(bytes([2]))
    print("Q3 Response:\n" + conn.recvall().decode('latin1'))

    conn.close()

if __name__ == '__main__':
    main()
```

- I ran the script and retrieved the flag: **flag{str1PP3d_B1N4R135_4r3_S0o0_much_FUN!_b2554c5e1abb41db}**



```
┌──(kali㉿kali)-[~]
└─$ python3 cryptex.py
[+] Opening connection to offsec-chalbroker.osiris.cyber.nyu.edu on port 1273: Done
Q1 Response:
*The first chamber opened! There is some weird carved into                   the interior...
\x06\x00\x00\x00
The second phase requires a single input...
 >
Q2 Response:
Nice, the second chamber opened! Ok, the final level requires another single input...
 >
[+] Receiving all data: Done (237B)
[*] Closed connection to offsec-chalbroker.osiris.cyber.nyu.edu port 1273
Q3 Response:

The final chamber opened, but a flaw in the design
popped a vinegar vial which started to eat away at the papyrus
scroll inside. You hold it up, trying to decipher the text...  flag{str1PP3d_B1N4R135_4r3_S0o0_much_FUN!_b2554c5e1abb41db}
```