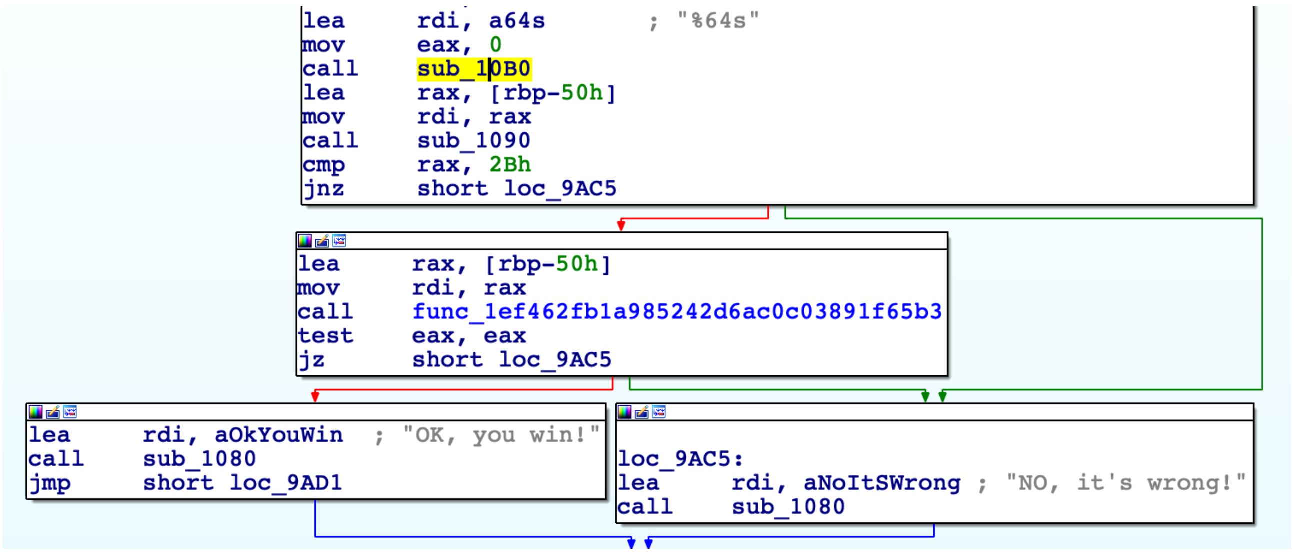
writeup - bamboofox2021

Flag Checker Revenge

While the contest was running, I solved this challenge with worst way which I manually found out every element of original flag. It's so stupid and it took me about some hours. After studying about **angr** framework and reading some examples using it, I solved this challenge again by using **angr.**
The source file of this challenge can be found at here.

Getting started

- Open **task** file with IDA tool, I recognized that:
    - The length of satisfied input is 0x2b(43)



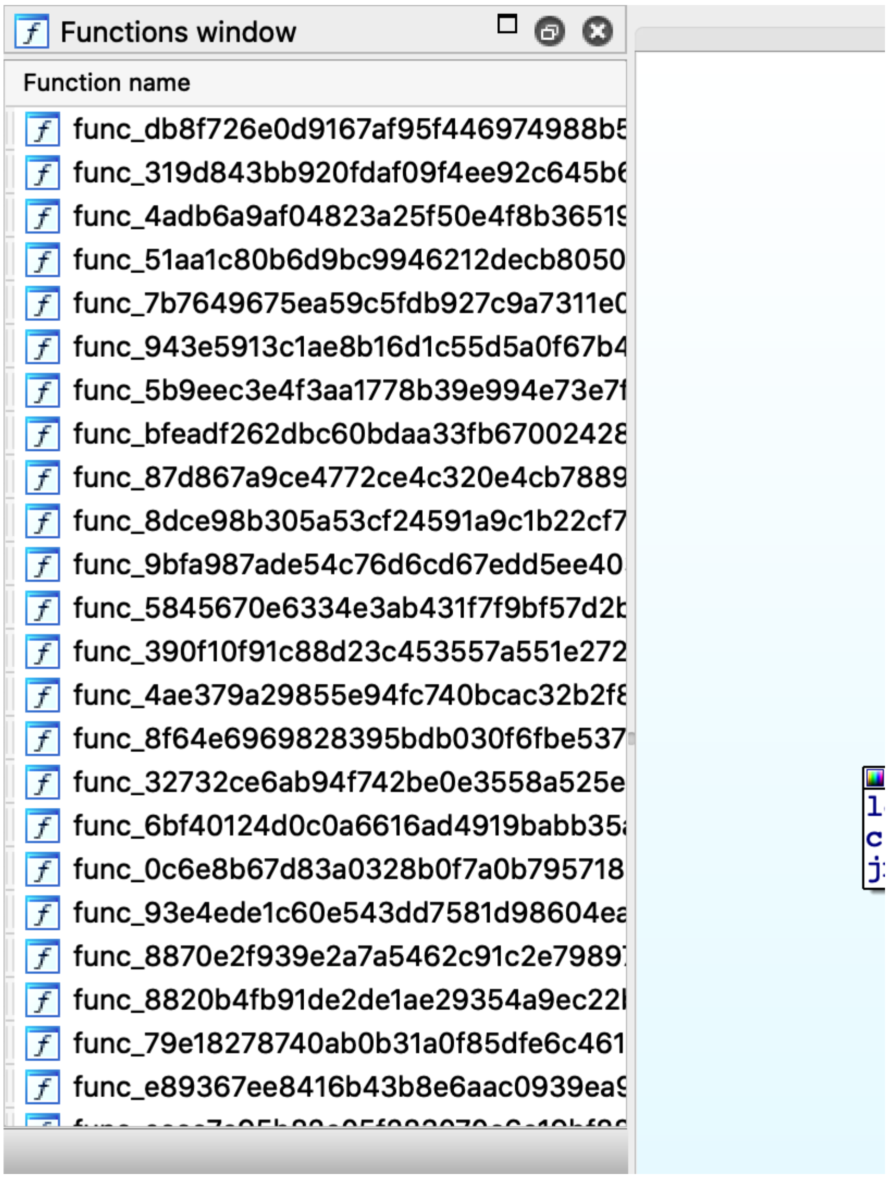
```
lea     rdi, a64s       ; "%64s"
mov     eax, 0
call    sub_10B0
lea     rax, [rbp-50h]
mov     rdi, rax
call    sub_1090
cmp     rax, 2Bh
jnz     short loc_9AC5
```

```
lea     rax, [rbp-50h]
mov     rdi, rax
call    func_1ef462fb1a985242d6ac0c03891f65b3
test    eax, eax
jz      short loc_9AC5
```

```
lea     rdi, aOkYouWin   ; "OK, you win!"
call    sub_1080
jmp     short loc_9AD1
```

```
loc_9AC5:
lea     rdi, aNoItSWrong ; "NO, it's wrong!"
call    sub_1080
```

-
    - there are so many checking functions which examine if our input satisfied or not:



Functions window

Function name

```
func_db8f726e0d9167af95f446974988b5
func_319d843bb920fdaf09f4ee92c645b6
func_4adb6a9af04823a25f50e4f8b3651
func_51aa1c80b6d9bc9946212decb8050
func_7b7649675ea59c5fdb927c9a7311e0
func_943e5913c1ae8b16d1c55d5a0f67b4
func_5b9eec3e4f3aa1778b39e994e73e7f
func_bfeadf262dbc60bdaa33fb67002428
func_87d867a9ce4772ce4c320e4cb7889
func_8dce98b305a53cf24591a9c1b22cf7
func_9bfa987ade54c76d6cd67edd5ee40
func_5845670e6334e3ab431f7f9bf57d2b
func_390f10f91c88d23c453557a551e272
func_4ae379a29855e94fc740bcac32b2f8
func_8f64e6969828395bdb030f6fbe537
func_32732ce6ab94f742be0e3558a525e
func_6bf40124d0c0a6616ad4919babb35
func_0c6e8b67d83a0328b0f7a0b795718
func_93e4ede1c60e543dd7581d98604ea
func_8870e2f939e2a7a5462c91c2e7989
func_8820b4fb91de2de1ae29354a9ec22
func_79e18278740ab0b31a0f85dfe6c461
func_e89367ee8416b43b8e6aac0939ea
```

- That time, I think about z3 and angr. I used to z3 solver, but with the big number of checking functions, using the z3 tool is impossible. I also haven't angr framework before, so I decided to solve challenge manually. It's successful but it's the most worst way.

Using *angr* framework

> A cool thing in ***angr*** is that symbolic execution. Say simply, tools of symbolic execution try to find all inputs of executable file which support for finding out the special output of that binary file.

> So, using ***angr*** in this challenge, it helps us generate the most appropriate input to achieve executable branch such as "*OK, you win!*"

- angr finds out the suitable inputs based on the address of instruction we want to jump to. They usually display *good* and *bad* address(bad address should not avoided).
- Additionally, we also specify the starting address and the position of flag in memory is in simulation-time
- Finally, each character of flag range from 32 to 127(in ASCII), so I add two constraints to solver. Below this is solved-script of this challenge:

solve_FlagCheckerRevenge.py

```python
import angr

p       = angr.Project("./task")

start   = 0x00009a95 + 0x400000
good   = 0x009AB7 + 0x400000
bad    = 0x009AC5 + 0x400000
st     = p.factory.blank_state(addr=start)

#specify the flag and its storing address in memory
st.regs.rbp = st.regs.rsp
st.regs.rsp = st.regs.rbp - 0x50
flag = st.solver.BVS("flag", 0x2B * 8)
st.memory.store(st.regs.rbp - 0x50, flag)

#each element of flag range in (32, 127)
for i in range(0x2B):
  char = (flag >> (8 * i)) & 0xFF
  st.solver.add(0x20 < char)
  st.solver.add(char < 0x7f)

#simulate execuatable file and find out the suitable input
sm = p.factory.simgr(st)
sm.explore(find=good, avoid=bad)
if sm.found:
  solution = sm.found[0].solver.eval(flag)
  flag_hex_str = hex(solution)[2:]
  for i in range(0, len(flag_hex_str), 2):
    print(chr(int(flag_hex_str[i:i+2], 16)), end='')
print()
```