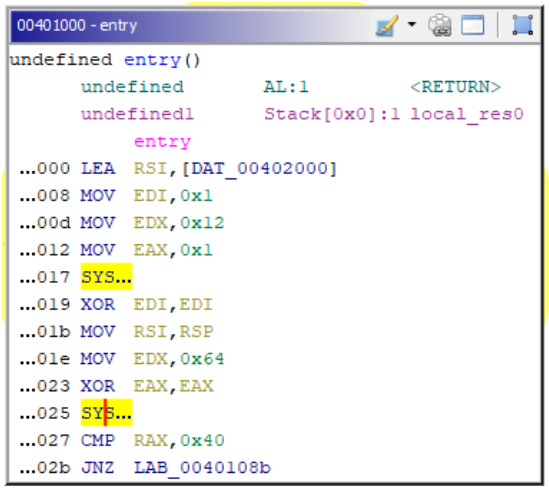
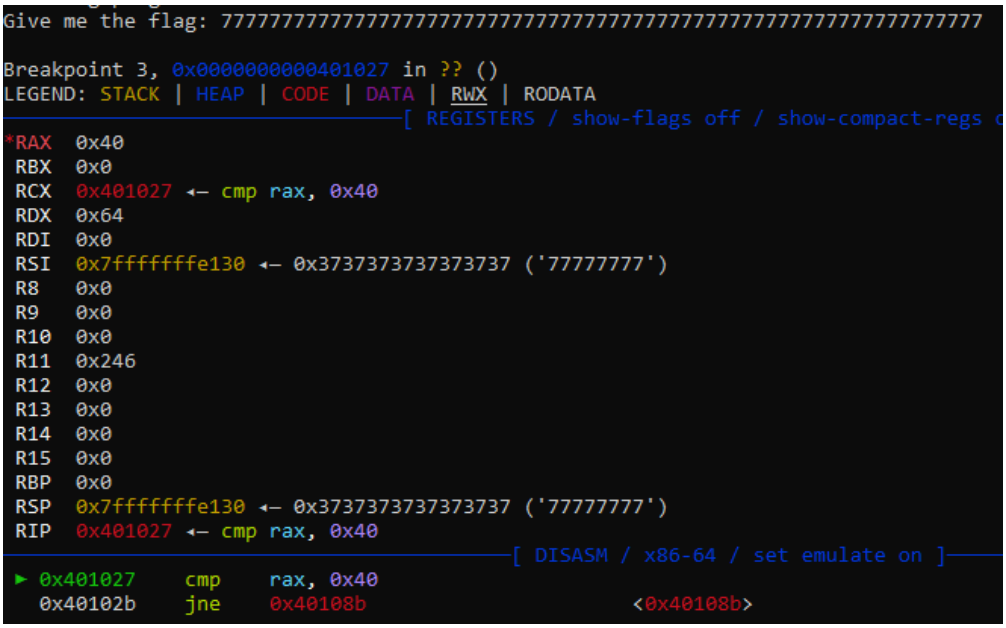
switcharoo

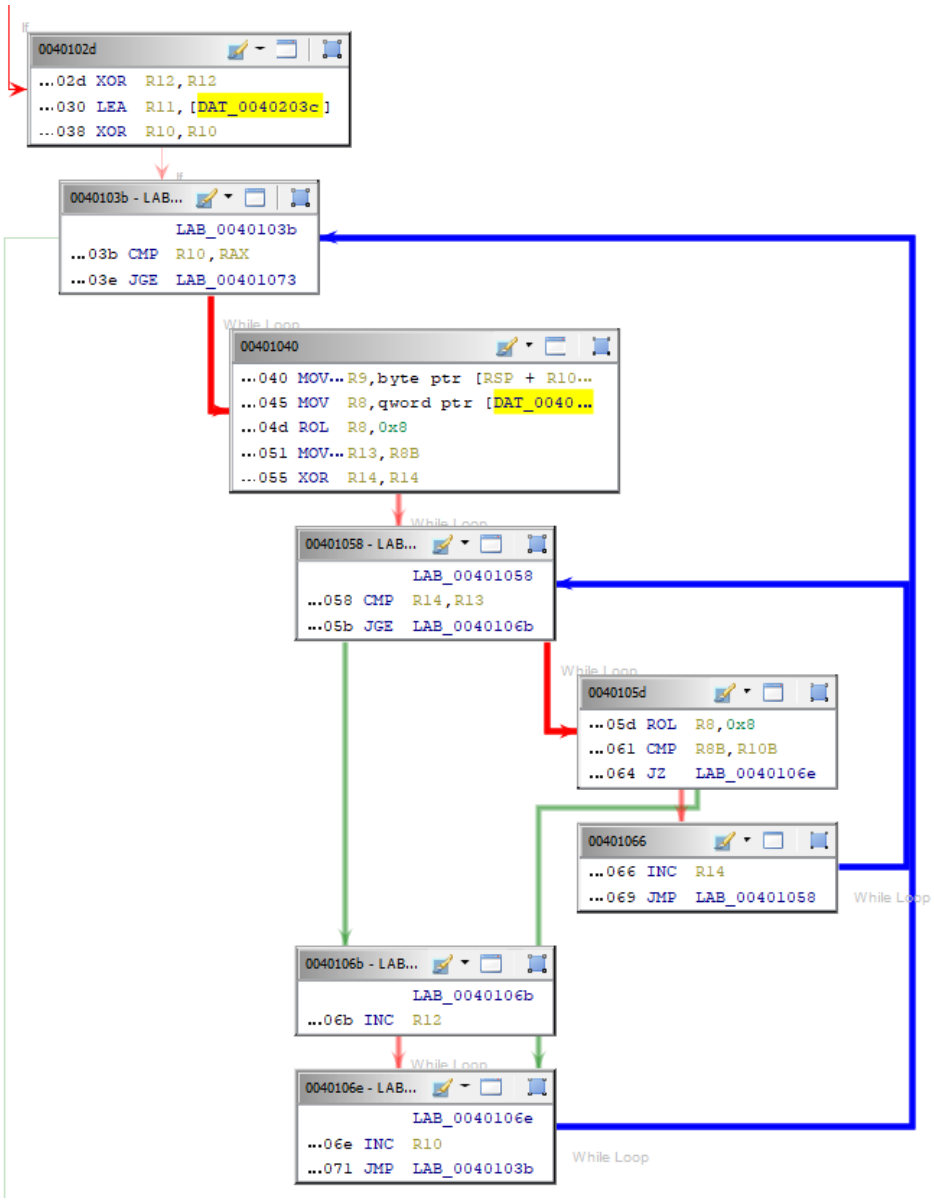
Let's look at the disassembly code.



The first syscall prints the prompt "Give me the flag:"

The second syscall reads in a user input, and right after it, it checks if the user input is of length 0x40 (decimal 64). As shown below when We supplied an input with the right length.





After passing the string length check, we enter the main section of the application, which can be broken down into an outer loop, and an inner loop. The overall logic here checks our user input character by character against bytes stored in the .data section, with DAT_0040203c being the offset.

R10 is the outer loop counter, which increments for every new character comparison.
R9 stores a character from our input string, which is the character we want to compare with.

```
00401045 4e 8b 04      MOV       R8,qword ptr [DAT_0040203c + R9*0x8]      = 06h
         cd 3c 20
         40 00
```

R8 stores a character from the .data section of the binary, which varies according to R9.
After storing the character, R8 performs a rotate left operation (ROL) of 8 bits.
R13 then stores the first lower byte of R8.

Now, let's try to dissect the inner loop.

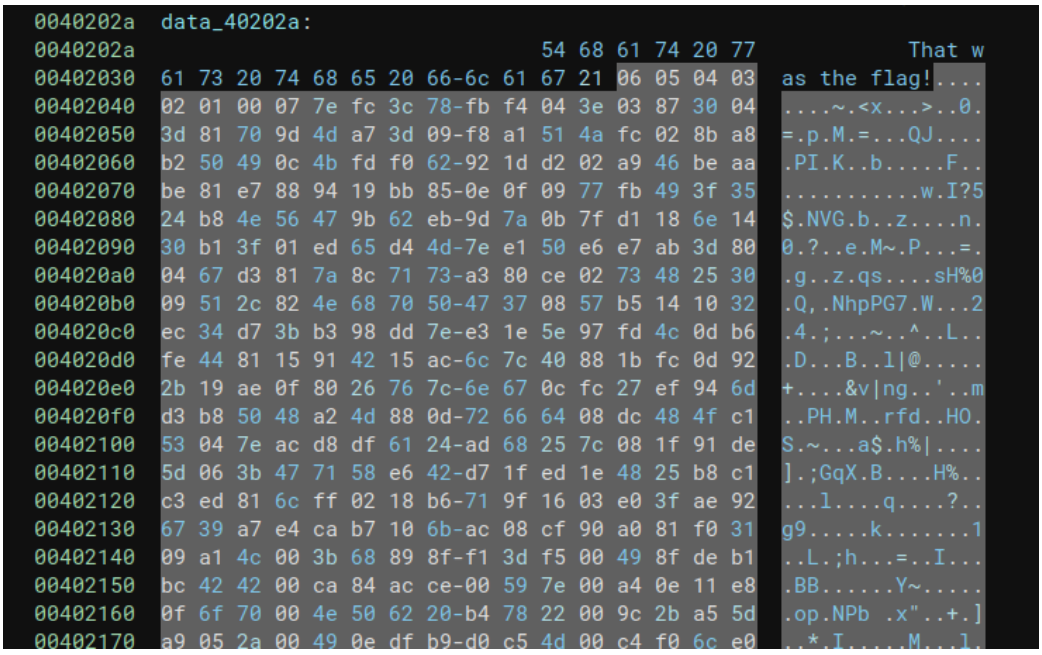
R14 is the inner loop counter, which breaks back out to the outer loop if it is >= R13.
If this happens, it means that our flag is incorrect since R12 is the test condition and it must not be incremented.

OTHERWISE, R8 performs a ROL again, and we go into the continuous checking for a match for the lower byte of R8 and R10. R14 gets incremented by 1 after every unsuccessful check and this happens as long as R14 < R13.

If we have a match, R10 is incremented by 1 and the process starts over again with the next user input character.

The main takeaway here is that we should never increment R12, as that would mean the string comparison failed.

The flag is said to be printable ASCII characters from 0x21 through 0x7E. We can most probably loop through the ASCII table for each character comparison and see what hits.



In Binary Ninja, we can navigate to the to 0x0040203c and copy over the byte elements as an array.

```python
        0xf6, 0x1a, 0x04, 0x04, 0x5d, 0xc8, 0xca, 0x00, 0x05, 0xba, 0x8e, 0x89, 0xaf, 0x41, 0x05, 0x02,
        0x95, 0x10, 0x67, 0x3d, 0x26, 0xb0, 0x2c, 0x00, 0x43, 0x36, 0xac, 0xe7, 0xff, 0x42, 0x3e, 0x02,
        0x87, 0x60, 0x09, 0x4e, 0x62, 0xf3, 0x19, 0x00, 0xec, 0x88, 0xa7, 0x5a, 0x99, 0xd3, 0xf8, 0x26]


def func():
    i = 0
    rax = 0
    r8 = 0
    tmp = 0

    while 1:
        for char in range(0x21, 0x7E):  # loop through printable ASCII chars
            i = 0

            r8 = arr[char*8:char*8+8]
            r8 = r8[::-1]  # little endian
            # ROL operation
            tmp = r8.pop(0)
            r8.append(tmp)

            r14 = 0
            r13 = r8[len(r8)-1]  # low byte from r8

            while 1:
                if r14 >= r13:
                    i += 1
                    break

                # ROL operation
                tmp = r8.pop(0)
                r8.append(tmp)

                if r8[len(r8)-1] == (rax & 0xff):
                    print(chr(char))
                    rax += 1
                else:
                    r14 += 1
    return

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

The script pretty much follows the main logic of the assembly code, where we printed out the ASCII character when R8b and R10b matches in the inner loop.

```
┌──(kali㉿kali)-[~/NYU_OffSec/LACTF/switcheroo]
└─$ python3 exploit.py
^Clactf{4223M8LY_5W17Ch_57473M3n75_4r3_7h3_4850LU73_8357_u+1f60a}Traceback (most recent call last):
  File "/home/kali/NYU_OffSec/LACTF/switcheroo/exploit.py", line 106, in <module>
    func()
  File "/home/kali/NYU_OffSec/LACTF/switcheroo/exploit.py", line 84, in func
    r8.append(tmp)
KeyboardInterrupt
```

**ignore other output.