

switcharoo

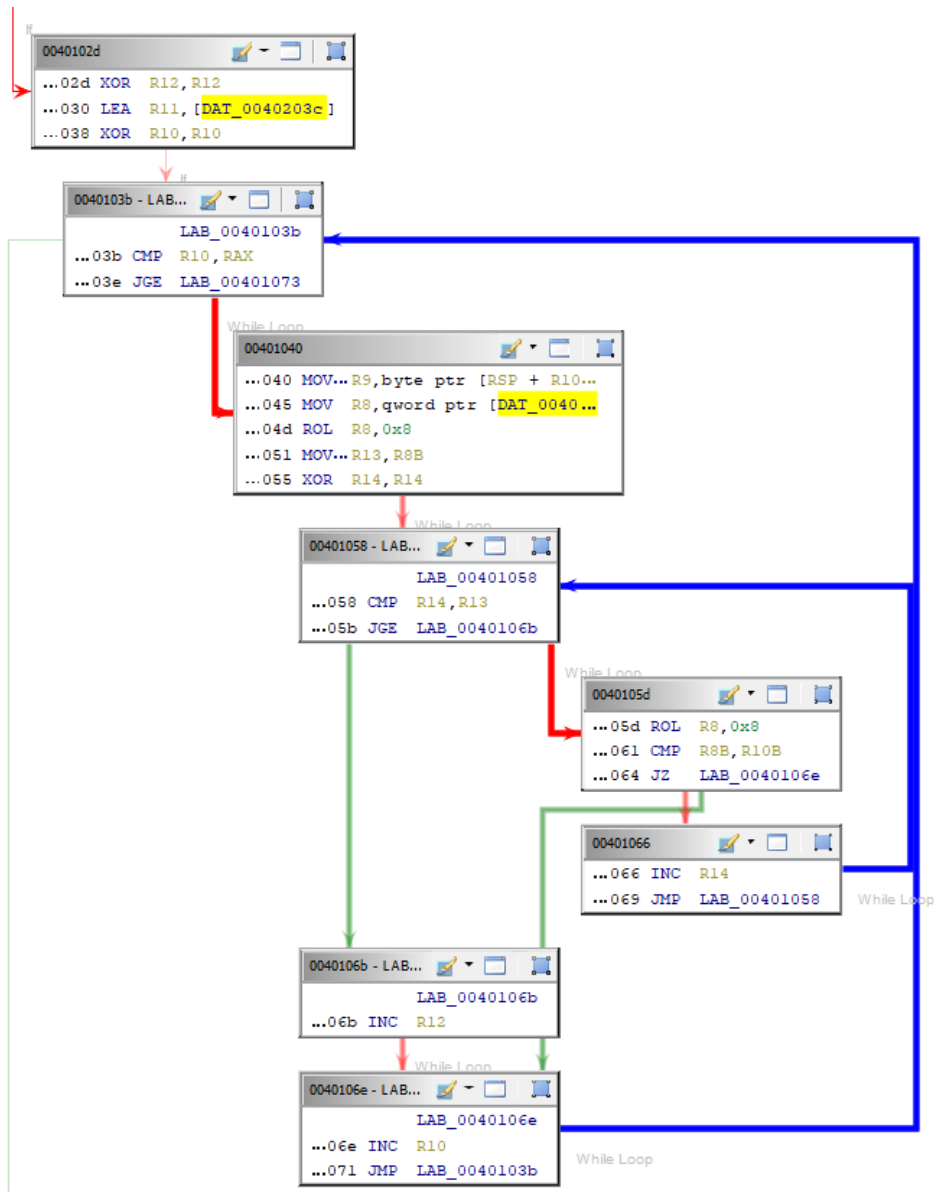
Let's look at the disassembly code.

```
00401000 -entry
undefined entry()
        undefined          AL:1          <RETURN>
        undefinedl         Stack[0x0]:1 local_res0
        entry
...000 LEA RSI,[DAT_00402000]
...008 MOV EDI,0x1
...00d MOV EDX,0x12
...012 MOV EAX,0x1
...017 SYS...
...019 XOR EDI,EDI
...01b MOV RSI,RSP
...01e MOV EDX,0x64
...023 XOR EAX,EAX
...025 SYS...
...027 CMP RAX,0x40
...02b JNZ LAB_0040108b
```

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.

[illegible]



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 \geq 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.

```
0040202a  data_40202a:
0040202a          54 68 61 74 20 77          That w
00402030  61 73 20 74 68 65 20 66-6c 61 67 21 06 05 04 03  as the flag!...
00402040  02 01 00 07 7e fc 3c 78-fb f4 04 3e 03 87 30 04  ....~.<x...>..0.
00402050  3d 81 70 9d 4d a7 3d 09-f8 a1 51 4a fc 02 8b a8  =.p.M.=...QJ....
00402060  b2 50 49 0c 4b fd f0 62-92 1d d2 02 a9 46 be aa  .PI.K..b.....F..
00402070  be 81 e7 88 94 19 bb 85-0e 0f 09 77 fb 49 3f 35  .....w.I?5
00402080  24 b8 4e 56 47 9b 62 eb-9d 7a 0b 7f d1 18 6e 14  $.NVG.b..z....n.
00402090  30 b1 3f 01 ed 65 d4 4d-7e e1 50 e6 e7 ab 3d 80  0.?..e.M~.P...=.
004020a0  04 67 d3 81 7a 8c 71 73-a3 80 ce 02 73 48 25 30  .g..z.qs....sH%0
004020b0  09 51 2c 82 4e 68 70 50-47 37 08 57 b5 14 10 32  .Q,.Nhpg7.W...2
004020c0  ec 34 d7 3b b3 98 dd 7e-e3 1e 5e 97 fd 4c 0d b6  .4.;...~..^..L..
004020d0  fe 44 81 15 91 42 15 ac-6c 7c 40 88 1b fc 0d 92  .D...B..l|@.....
004020e0  2b 19 ae 0f 80 26 76 7c-6e 67 0c fc 27 ef 94 6d  +...&v|ng...'..m
004020f0  d3 b8 50 48 a2 4d 88 0d-72 66 64 08 dc 48 4f c1  ..PH.M..rfd..HO.
00402100  53 04 7e ac d8 df 61 24-ad 68 25 7c 08 1f 91 de  S.~...a$.h%|....
00402110  5d 06 3b 47 71 58 e6 42-d7 1f ed 1e 48 25 b8 c1  ].;GqX.B....H%...
00402120  c3 ed 81 6c ff 02 18 b6-71 9f 16 03 e0 3f ae 92  ...l....q....?..
00402130  67 39 a7 e4 ca b7 10 6b-ac 08 cf 90 a0 81 f0 31  g9....k.....1
00402140  09 a1 4c 00 3b 68 89 8f-f1 3d f5 00 49 8f de b1  ..L.;h...=.I...
00402150  bc 42 42 00 ca 84 ac ce-00 59 7e 00 a4 0e 11 e8  .BB.....Y~.....
00402160  0f 6f 70 00 4e 50 62 20-b4 78 22 00 9c 2b a5 5d  .op.NPb .x"...+.]
00402170  a9 05 2a 00 49 0e df b9-d0 c5 4d 00 c4 f0 6c e0  ...*.I.....M....l..
```

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

```

65     0xf6, 0x1a, 0x04, 0x04, 0x5d, 0xc8, 0xca, 0x00, 0x05, 0xba, 0x8e, 0x89, 0xaf, 0x41, 0x05, 0x02,
66     0x95, 0x10, 0x67, 0x3d, 0x26, 0xb0, 0x2c, 0x00, 0x43, 0x36, 0xac, 0xe7, 0xff, 0x42, 0x3e, 0x02,
67     0x87, 0x60, 0x09, 0x4e, 0x62, 0xf3, 0x19, 0x00, 0xec, 0x88, 0xa7, 0x5a, 0x99, 0xd3, 0xf8, 0x26]
68
69
70 def func():
71     i = 0
72     rax = 0
73     r8 = 0
74     tmp = 0
75
76     while 1:
77         for char in range(0x21, 0x7E): # loop through printable ASCII chars
78             i = 0
79
80             r8 = arr[char*8:char*8+8]
81             r8 = r8[::-1] # little endian
82             # ROL operation
83             tmp = r8.pop(0)
84             r8.append(tmp)
85
86             r14 = 0
87             r13 = r8[len(r8)-1] # low byte from r8
88
89             while 1:
90                 if r14 >= r13:
91                     i += 1
92                     break
93
94                 # ROL operation
95                 tmp = r8.pop(0)
96                 r8.append(tmp)
97
98                 if r8[len(r8)-1] == (rax & 0xff):
99                     print(chr(char))
100                     rax += 1
101                 else:
102                     r14 += 1
103
104     return
105 if __name__ == '__main__':
106     func()
107

```

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
^Cactf{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.