**Take Some Byte**

***Reverse***

**Challenge:**

I think some code is need some effort to read.

Attached - byte.txt

**Approach:**

I’m just gonna paste the whole content of the byte.txt file here first and divide it into several sections for easy reference.

Section - 1

15 0 LOAD\_FAST 0 (flag)

2 LOAD\_CONST 0 (None)

4 LOAD\_CONST 1 (6)

6 BUILD\_SLICE 2

8 BINARY\_SUBSCR

10 LOAD\_CONST 2 ('TCP1P{')

12 COMPARE\_OP 3 (!=)

14 POP\_JUMP\_IF\_FALSE 38

16 LOAD\_FAST 0 (flag)

18 LOAD\_CONST 3 (-1)

20 LOAD\_CONST 0 (None)

22 BUILD\_SLICE 2

24 BINARY\_SUBSCR

26 LOAD\_CONST 4 ('}')

28 COMPARE\_OP 3 (!=)

30 POP\_JUMP\_IF\_FALSE 38

Section - 2

16 32 LOAD\_GLOBAL 0 (oops)

34 CALL\_FUNCTION 0

36 POP\_TOP

Section - 3

18 >> 38 LOAD\_FAST 0 (flag)

40 LOAD\_CONST 1 (6)

42 LOAD\_CONST 5 (10)

44 BUILD\_SLICE 2

46 BINARY\_SUBSCR

48 LOAD\_CONST 6 ('byte')

50 COMPARE\_OP 2 (==)

52 POP\_JUMP\_IF\_FALSE 60

Section - 4

19 54 LOAD\_GLOBAL 1 (yeayy)

56 CALL\_FUNCTION 0

58 POP\_TOP

Section - 5

21 >> 60 LOAD\_FAST 0 (flag)

62 LOAD\_CONST 5 (10)

64 BINARY\_SUBSCR

66 POP\_JUMP\_IF\_FALSE 98

68 LOAD\_FAST 0 (flag)

70 LOAD\_CONST 7 (15)

72 BINARY\_SUBSCR

74 POP\_JUMP\_IF\_FALSE 98

76 LOAD\_FAST 0 (flag)

78 LOAD\_CONST 8 (18)

80 BINARY\_SUBSCR

82 LOAD\_GLOBAL 2 (chr)

84 LOAD\_CONST 9 (95)

86 CALL\_FUNCTION 1

88 COMPARE\_OP 3 (!=)

90 POP\_JUMP\_IF\_FALSE 98

Section - 6

22 92 LOAD\_GLOBAL 0 (oops)

94 CALL\_FUNCTION 0

96 POP\_TOP

Section - 7

24 >> 98 LOAD\_FAST 0 (flag)

100 LOAD\_CONST 10 (11)

102 LOAD\_CONST 7 (15)

104 BUILD\_SLICE 2

106 BINARY\_SUBSCR

108 LOAD\_CONST 11 ('code')

110 COMPARE\_OP 3 (!=)

112 POP\_JUMP\_IF\_FALSE 120

Section - 8

25 114 LOAD\_GLOBAL 0 (oops)

116 CALL\_FUNCTION 0

118 POP\_TOP

Section - 9

27 >> 120 LOAD\_FAST 0 (flag)

122 LOAD\_CONST 10 (11)

124 BINARY\_SUBSCR

126 LOAD\_FAST 0 (flag)

128 LOAD\_CONST 12 (1)

130 BINARY\_SUBSCR

132 LOAD\_METHOD 3 (lower)

134 CALL\_METHOD 0

136 COMPARE\_OP 2 (==)

138 POP\_JUMP\_IF\_FALSE 146

Section - 10

28 140 LOAD\_GLOBAL 1 (yeayy)

142 CALL\_FUNCTION 0

144 POP\_TOP

Section - 11

30 >> 146 LOAD\_FAST 0 (flag)

148 LOAD\_CONST 13 (12)

150 BINARY\_SUBSCR

152 LOAD\_GLOBAL 4 (ord)

154 LOAD\_FAST 0 (flag)

156 LOAD\_CONST 14 (20)

158 BINARY\_SUBSCR

160 CALL\_FUNCTION 1

162 LOAD\_CONST 1 (6)

164 BINARY\_SUBTRACT

166 COMPARE\_OP 2 (==)

168 POP\_JUMP\_IF\_FALSE 176

Section - 12

31 170 LOAD\_GLOBAL 1 (yeayy)

172 CALL\_FUNCTION 0

174 POP\_TOP

Section - 13

33 >> 176 LOAD\_GLOBAL 4 (ord)

178 LOAD\_FAST 0 (flag)

180 LOAD\_CONST 15 (16)

182 BINARY\_SUBSCR

184 CALL\_FUNCTION 1

186 LOAD\_CONST 16 (105)

188 COMPARE\_OP 3 (!=)

190 POP\_JUMP\_IF\_FALSE 214

192 LOAD\_GLOBAL 4 (ord)

194 LOAD\_FAST 0 (flag)

196 LOAD\_CONST 17 (17)

198 BINARY\_SUBSCR

200 CALL\_FUNCTION 1

202 LOAD\_CONST 18 (115)

204 COMPARE\_OP 3 (!=)

206 POP\_JUMP\_IF\_FALSE 214

Section - 14

34 208 LOAD\_GLOBAL 0 (oops)

210 CALL\_FUNCTION 0

212 POP\_TOP

Section - 15

36 >> 214 LOAD\_FAST 0 (flag)

216 LOAD\_CONST 19 (19)

218 BINARY\_SUBSCR

220 LOAD\_CONST 20 ('H')

222 COMPARE\_OP 3 (!=)

224 POP\_JUMP\_IF\_FALSE 232

Section - 16

37 226 LOAD\_GLOBAL 0 (oops)

228 CALL\_FUNCTION 0

230 POP\_TOP

Section - 17

39 >> 232 LOAD\_GLOBAL 4 (ord)

234 LOAD\_FAST 0 (flag)

236 LOAD\_CONST 14 (20)

238 BINARY\_SUBSCR

240 CALL\_FUNCTION 1

242 LOAD\_CONST 21 (117)

244 COMPARE\_OP 2 (==)

246 POP\_JUMP\_IF\_FALSE 254

Section - 18

40 248 LOAD\_GLOBAL 1 (yeayy)

250 CALL\_FUNCTION 0

252 POP\_TOP

Section - 19

42 >> 254 LOAD\_GLOBAL 4 (ord)

256 LOAD\_FAST 0 (flag)

258 LOAD\_CONST 22 (21)

260 BINARY\_SUBSCR

262 CALL\_FUNCTION 1

264 LOAD\_GLOBAL 4 (ord)

266 LOAD\_FAST 0 (flag)

268 LOAD\_CONST 23 (2)

270 BINARY\_SUBSCR

272 CALL\_FUNCTION 1

274 LOAD\_CONST 5 (10)

276 BINARY\_SUBTRACT

278 COMPARE\_OP 3 (!=)

280 EXTENDED\_ARG 1

282 POP\_JUMP\_IF\_FALSE 290

Section - 20

43 284 LOAD\_GLOBAL 0 (oops)

286 CALL\_FUNCTION 0

288 POP\_TOP

Section - 21

45 >> 290 LOAD\_FAST 0 (flag)

292 LOAD\_CONST 24 (22)

294 BINARY\_SUBSCR

296 LOAD\_FAST 0 (flag)

298 LOAD\_CONST 25 (0)

300 BINARY\_SUBSCR

302 LOAD\_METHOD 3 (lower)

304 CALL\_METHOD 0

306 COMPARE\_OP 3 (!=)

308 EXTENDED\_ARG 1

310 POP\_JUMP\_IF\_FALSE 318

Section - 22

46 312 LOAD\_GLOBAL 0 (oops)

314 CALL\_FUNCTION 0

316 POP\_TOP

Section - 23

48 >> 318 LOAD\_FAST 0 (flag)

320 LOAD\_CONST 24 (22)

322 BINARY\_SUBSCR

324 LOAD\_FAST 0 (flag)

326 LOAD\_CONST 26 (23)

328 BINARY\_SUBSCR

330 COMPARE\_OP 2 (==)

332 EXTENDED\_ARG 1

334 POP\_JUMP\_IF\_FALSE 342

Section - 24

49 336 LOAD\_GLOBAL 1 (yeayy)

338 CALL\_FUNCTION 0

340 POP\_TOP

>> 342 LOAD\_CONST 0 (None)

344 RETURN\_VALUE

Now, as we’ve divided the whole “thing” into 24 different sections, we’ll start to work on them individually. Upon a quick google search we get to know this is python bytecode which is basically generated by the compiler for easy machine level understanding.

No, we don’t need to learn bytecode in order to solve this question (not entirely) but we’ll see specific functions that are used in this given bytecode. CPython uses a stack-based virtual machine, so basically push and pop functions so we’re gonna work with that.

A quick understanding of this bytecode reveals that the code is basically performing some operations and in response generating some sort of output (take a quick peek and we’ll see “yeayy” and “oops” so I'm basically guessing it's for the right and the wrong conditions.)

Checking it further, we see that sections 2,4,6,8,10,12,14,16,18,20,22 and 24 have almost nothing in it except except the “yeayy” / “oops”.

Let’s understand Section - 1 briefly:

Line 1 0 LOAD\_FAST 0 (flag)

Line 2 2 LOAD\_CONST 0 (None)

Line 3 4 LOAD\_CONST 1 (6)

Line 4 6 BUILD\_SLICE 2

Line 5 8 BINARY\_SUBSCR

Line 6 10 LOAD\_CONST 2 ('TCP1P{')

Line 7 12 COMPARE\_OP 3 (!=)

Line 8 14 POP\_JUMP\_IF\_FALSE 38

Line 9 16 LOAD\_FAST 0 (flag)

Line 10 18 LOAD\_CONST 3 (-1)

Line 11 20 LOAD\_CONST 0 (None)

Line 12 22 BUILD\_SLICE 2

Line 13 24 BINARY\_SUBSCR

Line 14 26 LOAD\_CONST 4 ('}')

Line 15 28 COMPARE\_OP 3 (!=)

Line 16 30 POP\_JUMP\_IF\_FALSE 38

Line 1 basically tells code to lead the variable ‘flag’ onto the stack

Line 2 tells the stack to load the constant None onto the stack

Line 3 loads the constant ‘6’ onto the stack

Line 4 BUILD\_SLICE is basically used to create a slice, it’ll first pop the top 2 elements from the stack which are None and 6 and then create a slice out of it and then push that onto the stack. (if you don't understand what a slice is, check next line and you’ll get an idea)

Line 5 BINARY\_SUBSCR basically pops the top 2 elements of the stack which are the recently pushed slice and the flag (which we pushed at the starting) and then uses that slice on the flag, which is basically equivalent to flag[None:6] = flag[6] and then pushes this segment onto the stack.

Line 6 then pushes the string ‘TCP1P{‘ onto the stack and

Line 7 first pops the top elements of the stack (‘TCP1P{‘ and flag[6]) and compares them.

This is all we need to understand, the logic behind this is not that difficult, now try to implement this same logic on Line 9 to Line 14.

What those lines are basically checking is the last index of the flag to be “}”

Now, we’ll be using this similar logic on sections 3,5,7,9,11,13,15,17,19,21 and 23 to craft a flag

Flag till now: **TCP1P{}**

Section-3:

This section tells us the value of flag[6:10] to be ‘byte’, thus

updated flag: **TCP1P{byte}**

Section-5:

This section tells that the index value of 10,15 and 18 should be present in the flag and should be equal to char(95) = \_

Therefore,

updated flag: **TCP1P{byte\_????\_??\_}**

Section-7:

Similar to section 3, this section tells us that value of flag[11:15] = ‘code’

Updated flag: **TCP1P{byte\_code\_??\_}**

Section-9:

This section checks that flag[11] should be equal to lower(flag[1]), which if we check is true, so that’s that, no update to the flag

Section-11:

This section checks that ord(flag[12]) = ord(flag[20]) - 6

Since we already know flag[12], we can calculate flag[20] out to be ‘u’

Updated flag: **TCP1P{byte\_code\_??\_?u}**

Section-13:

This section checks ord(flag[16])=105 and ord(flag[17])=115, which are basically for the letters ‘i’ and ‘s’ respectively.

Updated flag: **TCP1P{byte\_code\_is\_?u}**

Section-15:

Checks flag[19] = ‘H’

Updated flag: **TCP1P{byte\_code\_is\_Hu}**

Section-17:

ord(flag[20]) = 117

Which we already know, so no update

Section-19:

ord(flag[21]) = ord(flag[2]), which tells us that flag[21] = ‘P’

Updated flag: **TCP1P{byte\_code\_is\_HuP}**

Section-21:

flag[22] = lower(flag[0]), which tells us that flag[22] = ‘t’

Updated flag: **TCP1P{byte\_code\_is\_HuPt}**

Section-23:

flag[22] = flag[23] = ‘t’

Updated flag: **TCP1P{byte\_code\_is\_HuPtt}**

There are no more sections left and honestly the last part of the flag doesn’t make much of a sense, but well, that is the flag.

**Flag: TCP1P{byte\_code\_is\_HuPtt}**

Congrats!!

Happy Hacking!