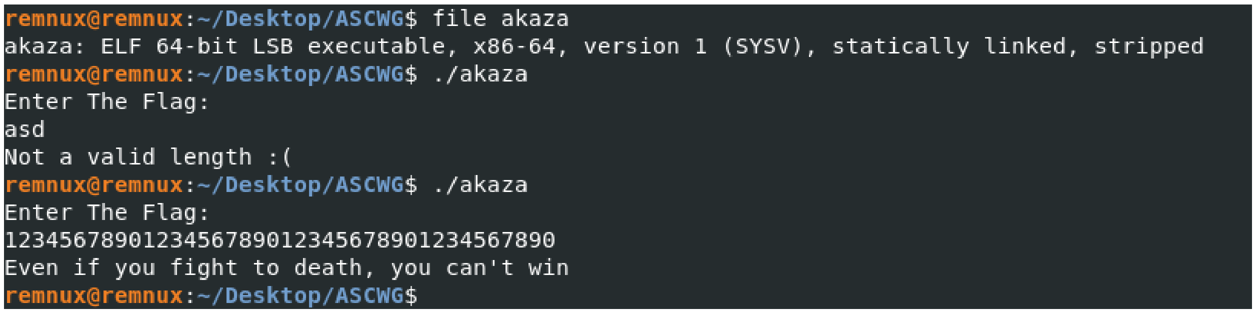
Details:

Author: **Yossef Zidan (@yossefzidann)**
Challenge Overview: The challenge goal is to find the correct password which is the flag the challenge consists of two stages the first one is a loader which is used to decrypt and load the other executable in memory which is a VM that does some checks against the flag.

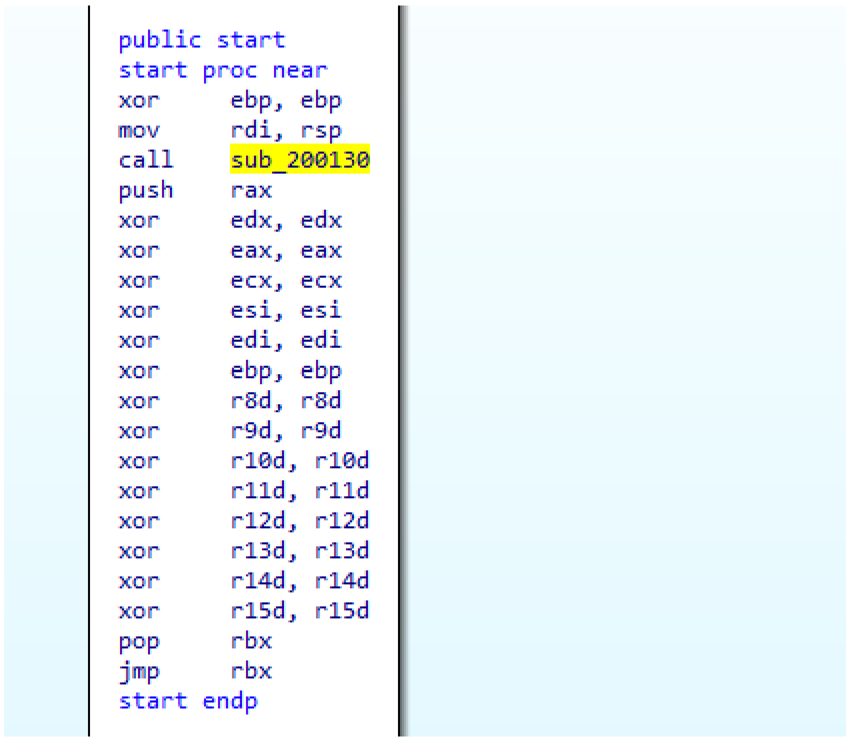
Step 1: Discovery



```
remnux@remnux:~/Desktop/ASCWG$ file akaza
akaza: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, stripped
remnux@remnux:~/Desktop/ASCWG$ ./akaza
Enter The Flag:
asd
Not a valid length :(
remnux@remnux:~/Desktop/ASCWG$ ./akaza
Enter The Flag:
1234567890123456789012345678901234567890
Even if you fight to death, you can't win
remnux@remnux:~/Desktop/ASCWG$
```

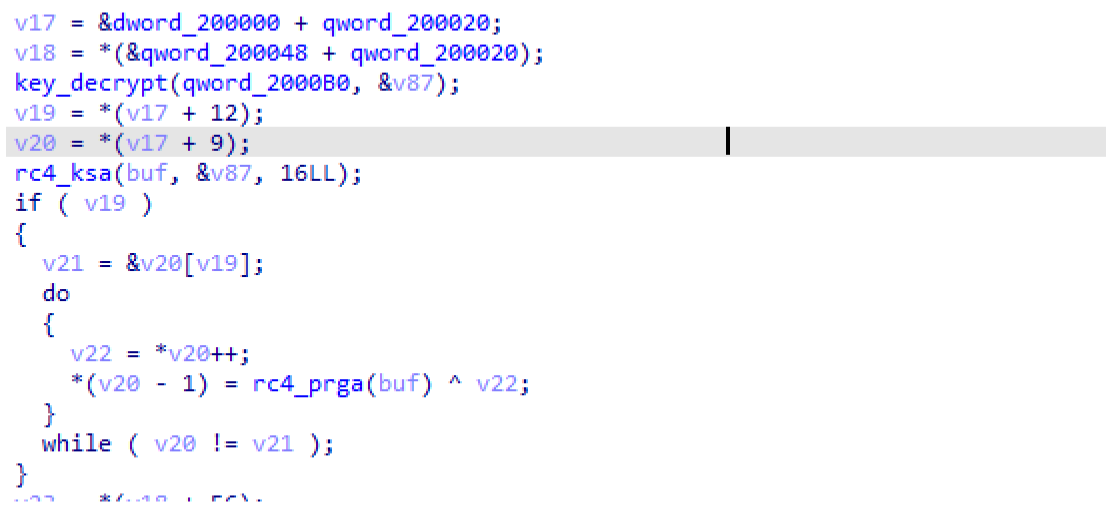
In this challenge, we are given one x64 elf file.

Step 2: Binary Analysis



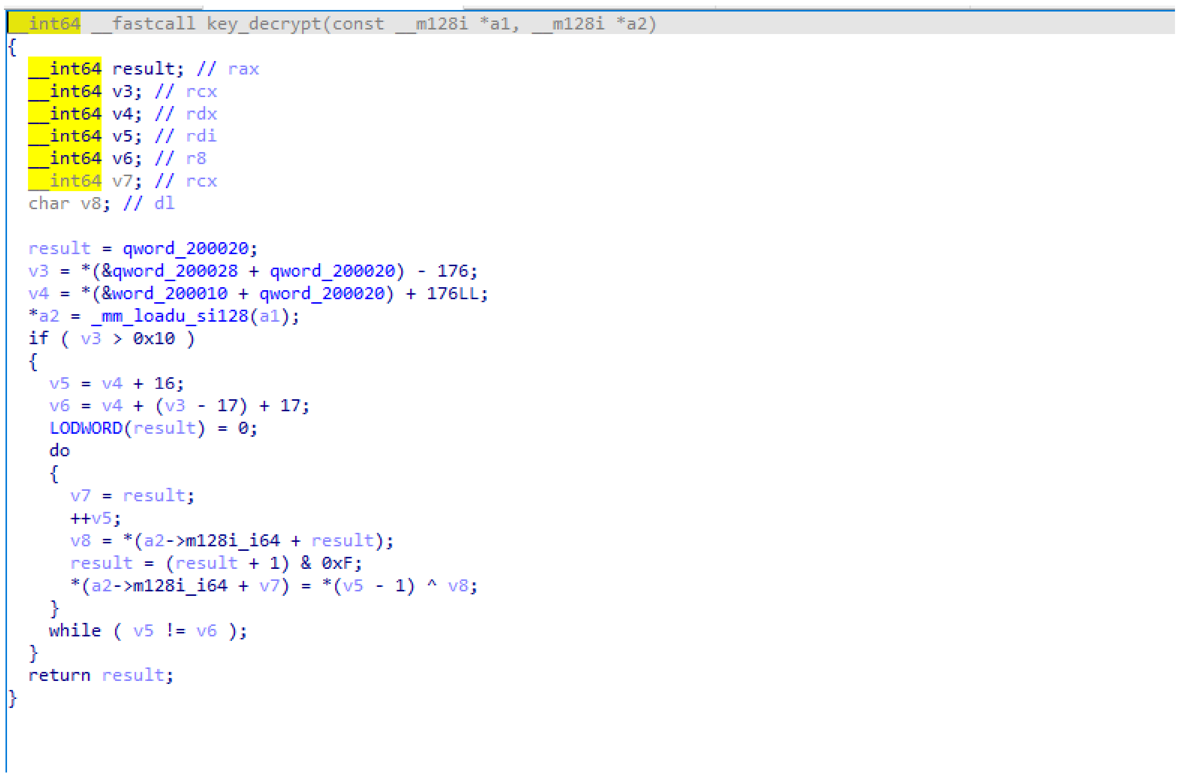
```
public start
start proc near
xor     ebp, ebp
mov     rdi, rsp
call    sub_200130
push    rax
xor     edx, edx
xor     eax, eax
xor     ecx, ecx
xor     esi, esi
xor     edi, edi
xor     ebp, ebp
xor     r8d, r8d
xor     r9d, r9d
xor     r10d, r10d
xor     r11d, r11d
xor     r12d, r12d
xor     r13d, r13d
xor     r14d, r14d
xor     r15d, r15d
pop     rbx
jmp     rbx
start endp
```

The loader calls the function sub_200130 which is used to decrypt the second stage executable and load it in memory and return the address of the OEP to execute the code of the second stage.



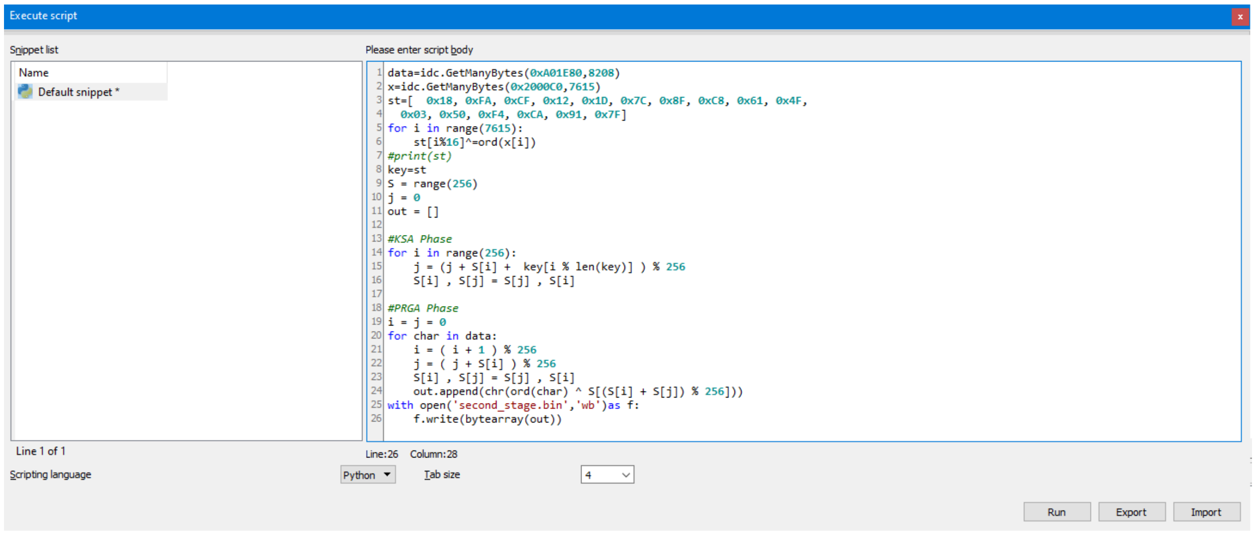
```
v17 = &dword_200000 + qword_200020;
v18 = *(&qword_200048 + qword_200020);
key_decrypt(qword_2000B0, &v87);
v19 = *(v17 + 12);
v20 = *(v17 + 9);
rc4_ksa(buf, &v87, 16LL);
if ( v19 )
{
  v21 = &v20[v19];
  do
  {
    v22 = *v20++;
    *(v20 - 1) = rc4_prga(buf) ^ v22;
  }
  while ( v20 != v21 );
}
```

First the rc4 key that will be used to decrypt the second stage will be decrypted.



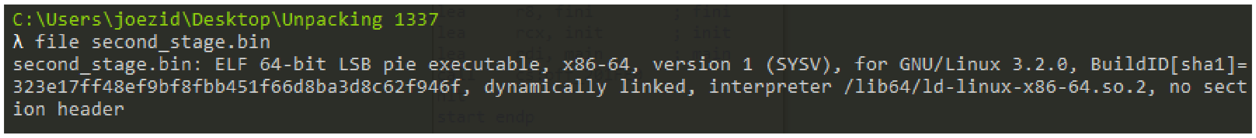
```
__int64 __fastcall key_decrypt(const __m128i *a1, __m128i *a2)
{
  __int64 result; // rax
  __int64 v3; // rcx
  __int64 v4; // rdx
  __int64 v5; // rdi
  __int64 v6; // r8
  __int64 v7; // rcx
  char v8; // dl

  result = qword_200020;
  v3 = *(&qword_200028 + qword_200020) - 176;
  v4 = *(&word_200010 + qword_200020) + 176LL;
  *a2 = _mm_loadu_si128(a1);
  if ( v3 > 0x10 )
  {
    v5 = v4 + 16;
    v6 = v4 + (v3 - 17) + 17;
    LODWORD(result) = 0;
    do
    {
      v7 = result;
      ++v5;
      v8 = *(a2->m128i_i64 + result);
      result = (result + 1) & 0xF;
      *(a2->m128i_i64 + v7) = *(v5 - 1) ^ v8;
    }
    while ( v5 != v6 );
  }
  return result;
}
```

The key will be decrypted by xoring the 16 byte key with all the bytes of the loader.



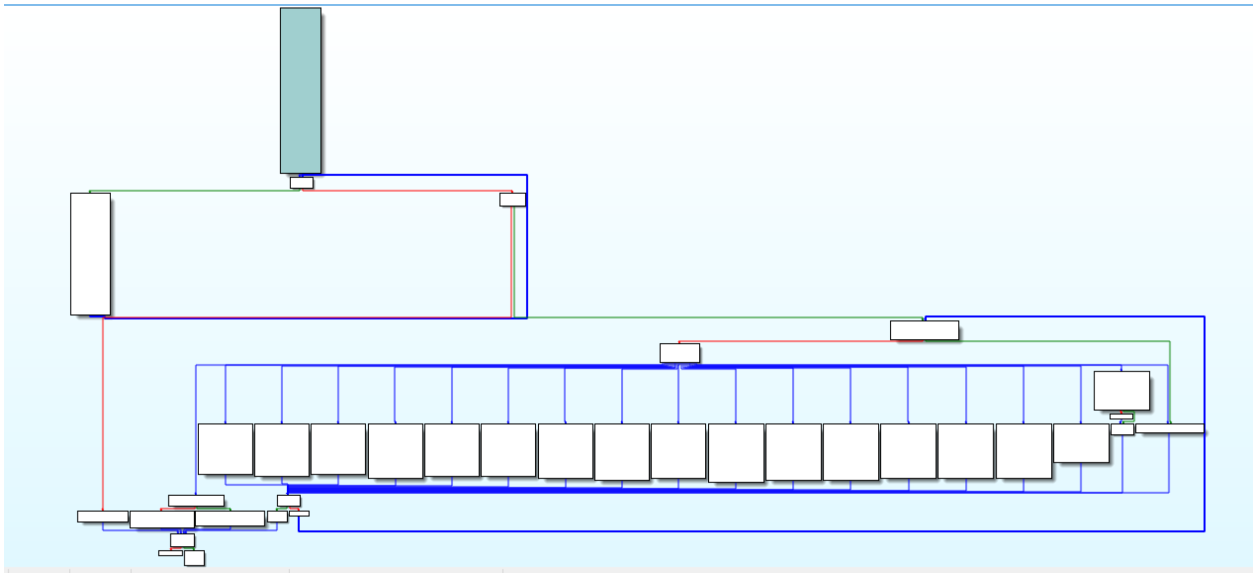
```python
data=idc.GetManyBytes(0xA01E80,8208)
x=idc.GetManyBytes(0x2000C0,7615)
st=[  0x18, 0xFA, 0xCF, 0x12, 0x1D, 0x7C, 0x8F, 0xC8, 0x61, 0x4F,
  0x03, 0x50, 0xF4, 0xCA, 0x91, 0x7F]
for i in range(7615):
    st[i%16]^=ord(x[i])
#print(st)
key=st
S = range(256)
j = 0
out = []

#KSA Phase
for i in range(256):
    j = (j + S[i] +  key[i % len(key)] ) % 256
    S[i] , S[j] = S[j] , S[i]

#PRGA Phase
i = j = 0
for char in data:
    i = ( i + 1 ) % 256
    j = ( j + S[i] ) % 256
    S[i] , S[j] = S[j] , S[i]
    out.append(chr(ord(char) ^ S[(S[i] + S[j]) % 256]))
with open('second_stage.bin','wb')as f:
    f.write(bytearray(out))
```

We can write a simple ida python script to decrypt the key then use it to decrypt the second stage executable.

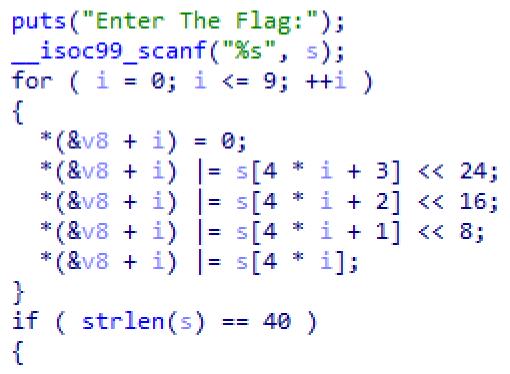


```
C:\Users\joezid\Desktop\Unpacking 1337
λ file second_stage.bin
second_stage.bin: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), for GNU/Linux 3.2.0, BuildID[sha1]=
323e17ff48ef9bf8fbb451f66d8ba3d8c62f946f, dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, no sect
ion header
```

Now we have the second stage executable we can start analyzing it.

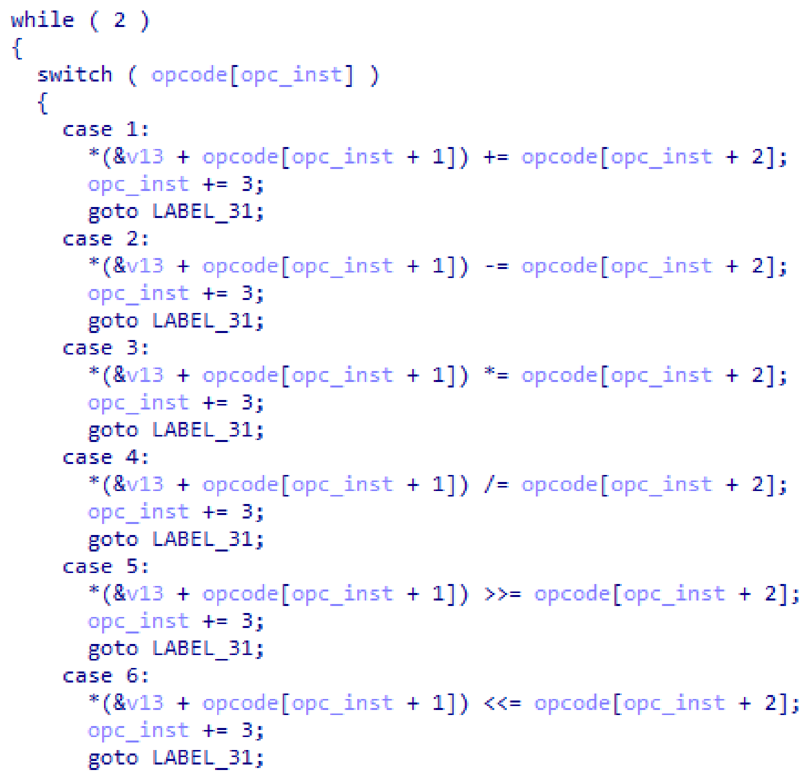


The code contains alot of branches.



```
puts("Enter The Flag:");
__isoc99_scanf("%s", s);
for ( i = 0; i <= 9; ++i )
{
  *(&v8 + i) = 0;
  *(&v8 + i) |= s[4 * i + 3] << 24;
  *(&v8 + i) |= s[4 * i + 2] << 16;
  *(&v8 + i) |= s[4 * i + 1] << 8;
  *(&v8 + i) |= s[4 * i];
}
if ( strlen(s) == 40 )
{
```

The program starts by getting the user input and splits it into blocks of 4 bytes and finally checks the length of the input if it equals 40.

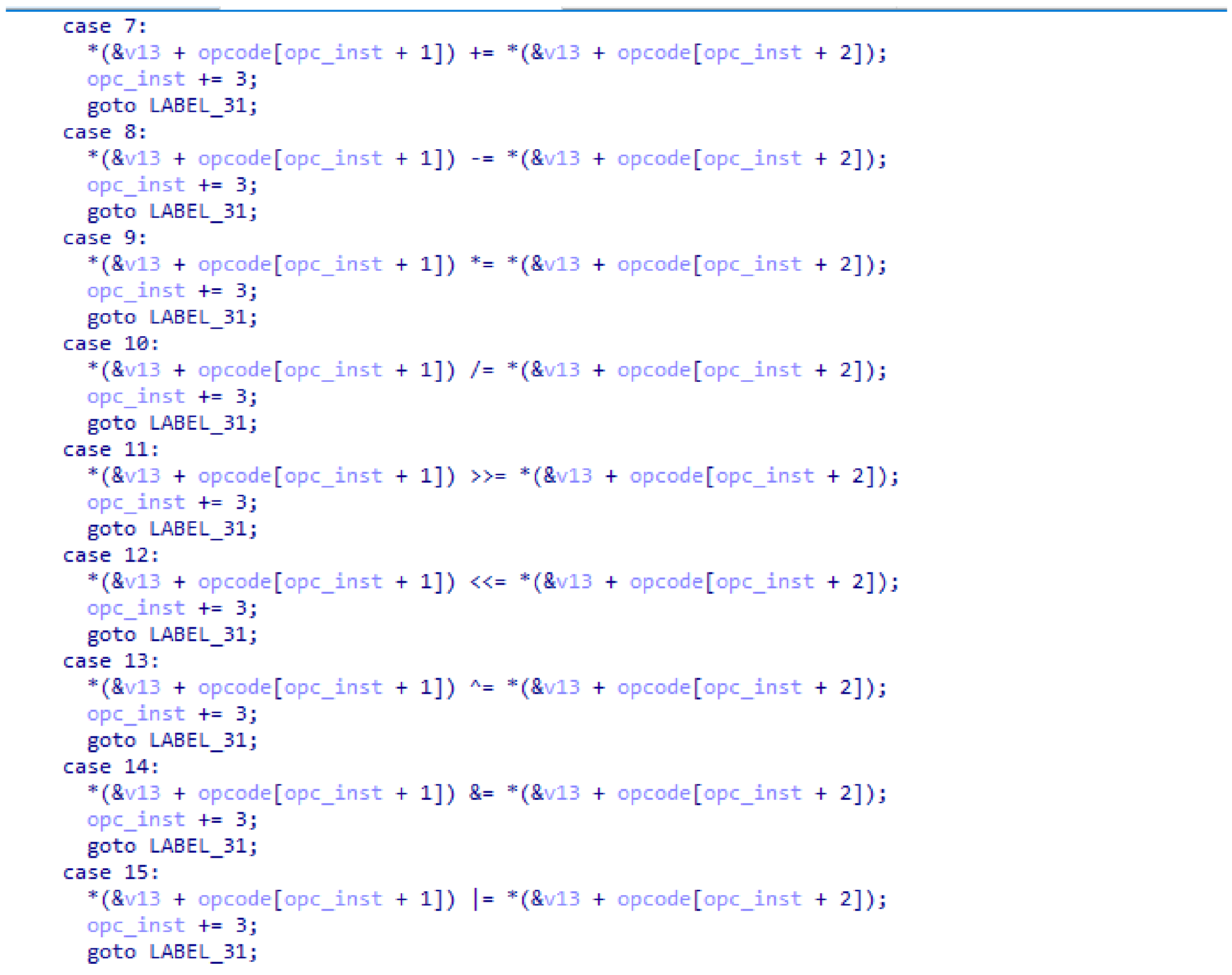


```
while ( 2 )
{
  switch ( opcode[opc_inst] )
  {
    case 1:
      *(&v13 + opcode[opc_inst + 1]) += opcode[opc_inst + 2];
      opc_inst += 3;
      goto LABEL_31;
    case 2:
      *(&v13 + opcode[opc_inst + 1]) -= opcode[opc_inst + 2];
      opc_inst += 3;
      goto LABEL_31;
    case 3:
      *(&v13 + opcode[opc_inst + 1]) *= opcode[opc_inst + 2];
      opc_inst += 3;
      goto LABEL_31;
    case 4:
      *(&v13 + opcode[opc_inst + 1]) /= opcode[opc_inst + 2];
      opc_inst += 3;
      goto LABEL_31;
    case 5:
      *(&v13 + opcode[opc_inst + 1]) >>= opcode[opc_inst + 2];
      opc_inst += 3;
      goto LABEL_31;
    case 6:
      *(&v13 + opcode[opc_inst + 1]) <<= opcode[opc_inst + 2];
      opc_inst += 3;
      goto LABEL_31;
```

Then we will start the vm code analysis we have 20 registers of size 32 bit and one instruction pointer and we have over all of 18 available operations.

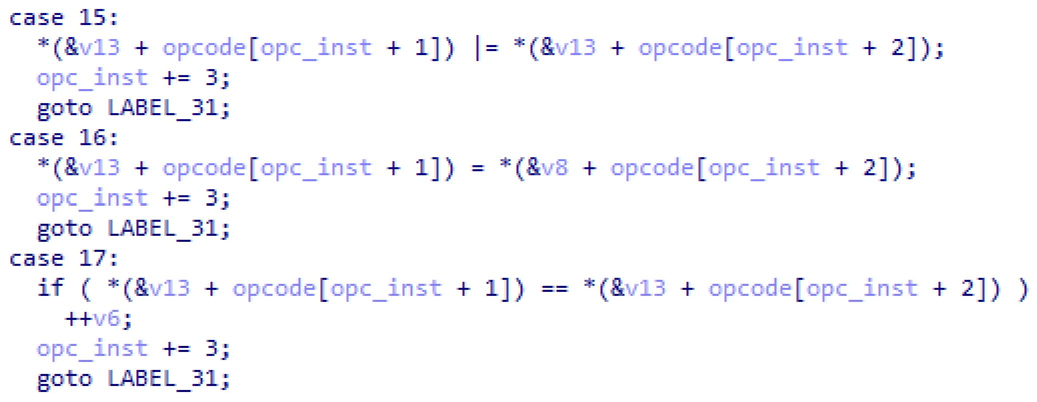
The first 6 operations are for direct arithmetic or logical operations with the passed value with the first byte of the opcode as the type of operation and second byte for register number and last byte is the hardcoded value.

So for example if the opcode is “\x01\x01\x66” the operation will be regs[0]+=0x66.



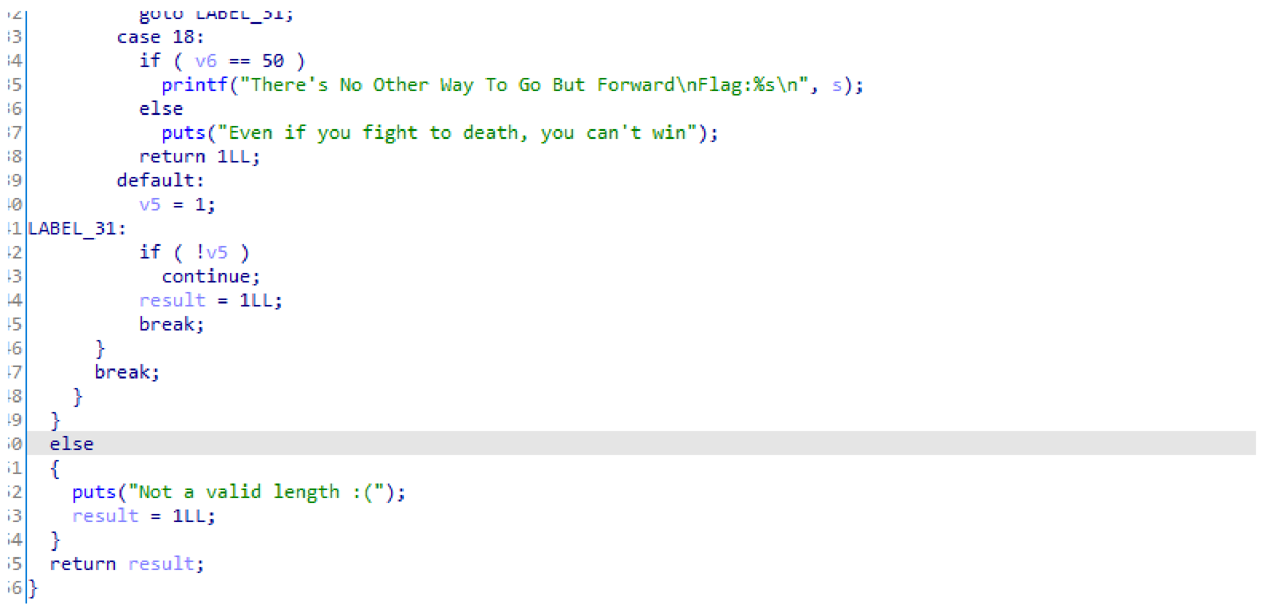
```
  case 7:
    *(&v13 + opcode[opc_inst + 1]) += *(&v13 + opcode[opc_inst + 2]);
    opc_inst += 3;
    goto LABEL_31;
  case 8:
    *(&v13 + opcode[opc_inst + 1]) -= *(&v13 + opcode[opc_inst + 2]);
    opc_inst += 3;
    goto LABEL_31;
  case 9:
    *(&v13 + opcode[opc_inst + 1]) *= *(&v13 + opcode[opc_inst + 2]);
    opc_inst += 3;
    goto LABEL_31;
  case 10:
    *(&v13 + opcode[opc_inst + 1]) /= *(&v13 + opcode[opc_inst + 2]);
    opc_inst += 3;
    goto LABEL_31;
  case 11:
    *(&v13 + opcode[opc_inst + 1]) >>= *(&v13 + opcode[opc_inst + 2]);
    opc_inst += 3;
    goto LABEL_31;
  case 12:
    *(&v13 + opcode[opc_inst + 1]) <<= *(&v13 + opcode[opc_inst + 2]);
    opc_inst += 3;
    goto LABEL_31;
  case 13:
    *(&v13 + opcode[opc_inst + 1]) ^= *(&v13 + opcode[opc_inst + 2]);
    opc_inst += 3;
    goto LABEL_31;
  case 14:
    *(&v13 + opcode[opc_inst + 1]) &= *(&v13 + opcode[opc_inst + 2]);
    opc_inst += 3;
    goto LABEL_31;
  case 15:
    *(&v13 + opcode[opc_inst + 1]) |= *(&v13 + opcode[opc_inst + 2]);
    opc_inst += 3;
    goto LABEL_31;
```

From operations 7 to 15 they are used for operation between registers so if the opcode is “\x07\x02\x03” the operation will be regs[2]+=regs[3].



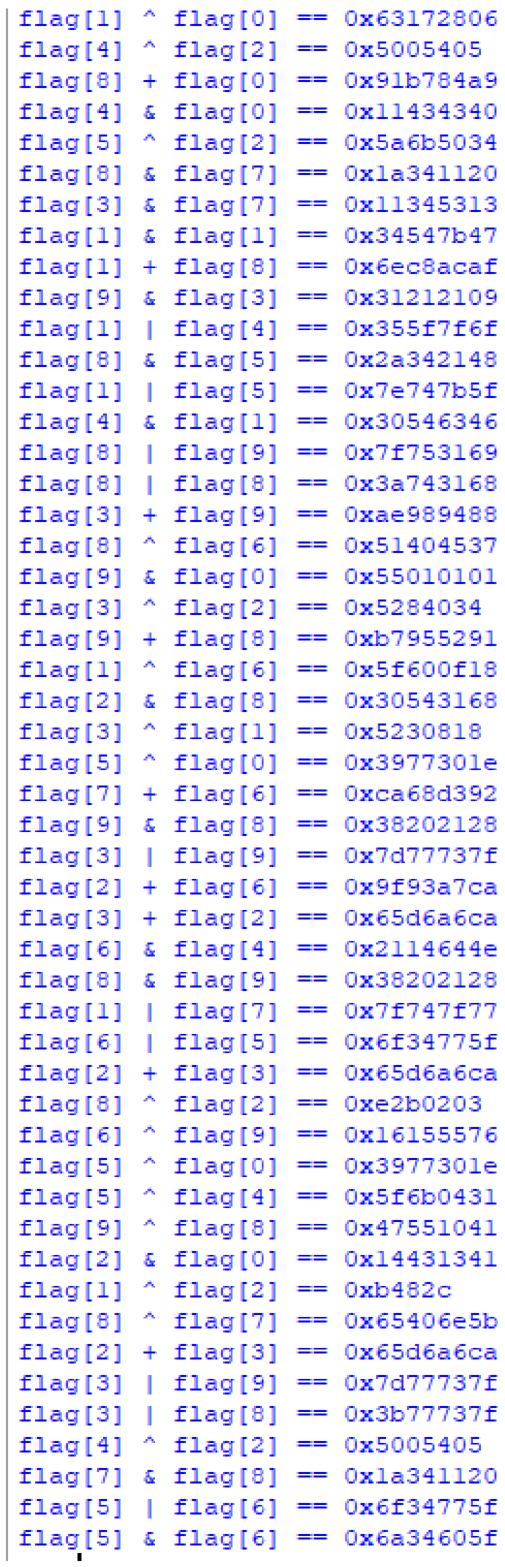
```
case 15:
  *(&v13 + opcode[opc_inst + 1]) |= *(&v13 + opcode[opc_inst + 2]);
  opc_inst += 3;
  goto LABEL_31;
case 16:
  *(&v13 + opcode[opc_inst + 1]) = *(&v8 + opcode[opc_inst + 2]);
  opc_inst += 3;
  goto LABEL_31;
case 17:
  if ( *(&v13 + opcode[opc_inst + 1]) == *(&v13 + opcode[opc_inst + 2]) )
    ++v6;
  opc_inst += 3;
  goto LABEL_31;
```

Operation number 16 will move one of the flag blocks to a register and operation number 17 will compare between to registers and if they are equal it will increment the value of v6.



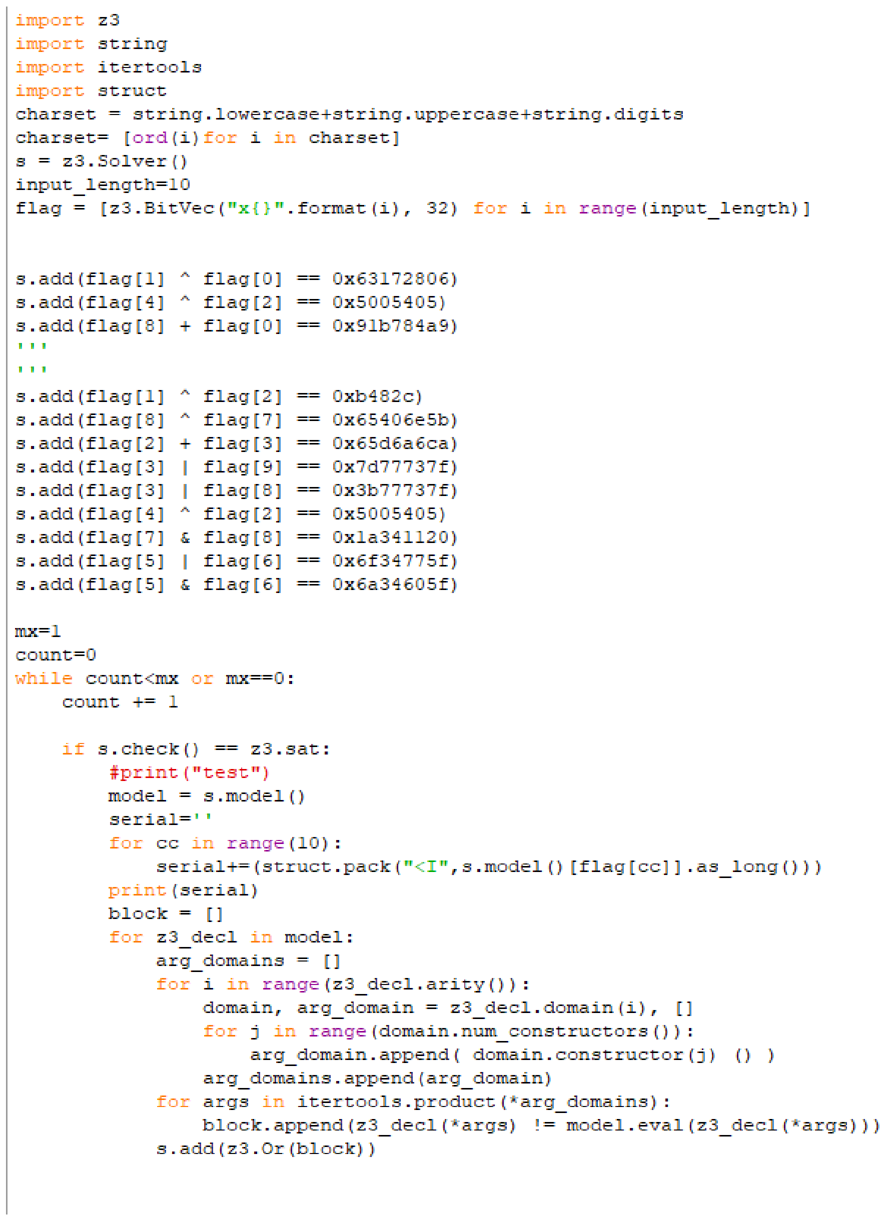
```
          goto LABEL_31;
        case 18:
          if ( v6 == 50 )
            printf("There's No Other Way To Go But Forward\nFlag:%s\n", s);
          else
            puts("Even if you fight to death, you can't win");
          return 1LL;
        default:
          v5 = 1;
LABEL_31:
          if ( !v5 )
            continue;
          result = 1LL;
          break;
      }
      break;
    }
  }
  else
  {
    puts("Not a valid length :(");
    result = 1LL;
  }
  return result;
}
```

The last operation is used to check if we passed all the 50 checks and if we pass all then we passed the correct flag.



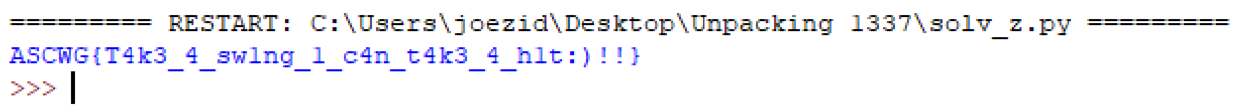
```
flag[1] ^ flag[0] == 0x63172806
flag[4] ^ flag[2] == 0x5005405
flag[8] + flag[0] == 0x91b784a9
flag[4] & flag[0] == 0x11434340
flag[5] ^ flag[2] == 0x5a6b5034
flag[8] & flag[7] == 0x1a341120
flag[3] & flag[7] == 0x11345313
flag[1] & flag[1] == 0x34547b47
flag[1] + flag[8] == 0x6ec8acaf
flag[9] & flag[3] == 0x31212109
flag[1] | flag[4] == 0x355f7f6f
flag[8] & flag[5] == 0x2a342148
flag[1] | flag[5] == 0x7e747b5f
flag[4] & flag[1] == 0x30546346
flag[8] | flag[9] == 0x7f753169
flag[8] | flag[8] == 0x3a743168
flag[3] + flag[9] == 0xae989488
flag[8] ^ flag[6] == 0x51404537
flag[9] & flag[0] == 0x55010101
flag[3] ^ flag[2] == 0x5284034
flag[9] + flag[8] == 0xb7955291
flag[1] ^ flag[6] == 0x5f600f18
flag[2] & flag[8] == 0x30543168
flag[3] ^ flag[1] == 0x5230818
flag[5] ^ flag[0] == 0x3977301e
flag[7] + flag[6] == 0xca68d392
flag[9] & flag[8] == 0x38202128
flag[3] | flag[9] == 0x7d77737f
flag[2] + flag[6] == 0x9f93a7ca
flag[3] + flag[2] == 0x65d6a6ca
flag[6] & flag[4] == 0x2114644e
flag[8] & flag[9] == 0x38202128
flag[1] | flag[7] == 0x7f747f77
flag[6] | flag[5] == 0x6f34775f
flag[2] + flag[3] == 0x65d6a6ca
flag[8] ^ flag[2] == 0xe2b0203
flag[6] ^ flag[9] == 0x16155576
flag[5] ^ flag[0] == 0x3977301e
flag[5] ^ flag[4] == 0x5f6b0431
flag[9] ^ flag[8] == 0x47551041
flag[2] & flag[0] == 0x14431341
flag[1] ^ flag[2] == 0xb482c
flag[8] ^ flag[7] == 0x65406e5b
flag[2] + flag[3] == 0x65d6a6ca
flag[3] | flag[9] == 0x7d77737f
flag[3] | flag[8] == 0x3b77737f
flag[4] ^ flag[2] == 0x5005405
flag[7] & flag[8] == 0x1a341120
flag[5] | flag[6] == 0x6f34775f
flag[5] & flag[6] == 0x6a34605f
```

Analyzing the vm we can see that we have to find the correct flag that satisfies all the equations we have so we can use any smt solver like z3 to find the flag.



```
import z3
import string
import itertools
import struct
charset = string.lowercase+string.uppercase+string.digits
charset= [ord(i)for i in charset]
s = z3.Solver()
input_length=10
flag = [z3.BitVec("x{}".format(i), 32) for i in range(input_length)]


s.add(flag[1] ^ flag[0] == 0x63172806)
s.add(flag[4] ^ flag[2] == 0x5005405)
s.add(flag[8] + flag[0] == 0x91b784a9)
'''
'''
s.add(flag[1] ^ flag[2] == 0xb482c)
s.add(flag[8] ^ flag[7] == 0x65406e5b)
s.add(flag[2] + flag[3] == 0x65d6a6ca)
s.add(flag[3] | flag[9] == 0x7d77737f)
s.add(flag[3] | flag[8] == 0x3b77737f)
s.add(flag[4] ^ flag[2] == 0x5005405)
s.add(flag[7] & flag[8] == 0x1a341120)
s.add(flag[5] | flag[6] == 0x6f34775f)
s.add(flag[5] & flag[6] == 0x6a34605f)

mx=1
count=0
while count<mx or mx==0:
    count += 1

    if s.check() == z3.sat:
        #print("test")
        model = s.model()
        serial=''
        for cc in range(10):
            serial+=(struct.pack("<I",s.model()[flag[cc]].as_long()))
        print(serial)
        block = []
        for z3_decl in model:
            arg_domains = []
            for i in range(z3_decl.arity()):
                domain, arg_domain = z3_decl.domain(i), []
                for j in range(domain.num_constructors()):
                    arg_domain.append( domain.constructor(j)  () )
                arg_domains.append(arg_domain)
            for args in itertools.product(*arg_domains):
                block.append(z3_decl(*args) != model.eval(z3_decl(*args)))
            s.add(z3.Or(block))
```

Running the code will give us the flag.



```
========= RESTART: C:\Users\joezid\Desktop\Unpacking 1337\solv_z.py =========
ASCWG{T4k3_4_sw1ng_1_c4n_t4k3_4_h1t:)!!}
>>> 
```

Flag: **ASCWG{T4k3_4_sw1ng_1_c4n_t4k3_4_h1t:)!!}**