SuperValidador - 400 pts - Reversing

La descripción del reto nos dice lo siguiente:

Es momento de algo serio, encuentra la llave y obten la flag!

Ademas de que una hint fue que **z3** podría ayudarnos. Basados en eso, para este reto decidimos abrir el binario usando **Ghidra** para poder ver un decompilado como el de **DnSpy**

```
24  _printf("Llave: ");
25  local_2c = (char *)_malloc(0xfe);
26  pcVar1 = _fgets(local_2c,0xfe,(FILE *)_iob_exref);
27  if (pcVar1 == (char *)0x0) {
28    _printf("PIRATA DETECTADO!!!!");
29    iVar2 = -1;
30  }
31  else {
32    local_30 = _mmmm((int)local_2c);
33    local_34 = 4;
34    local_38 = 0x13;
35    local_3c = 0x41d80000;
36    local_14 = 0x80;
37    iVar2 = local_14;
38    do {
39      local_14 = iVar2;
40      iVar2 = local_14 >> 1;
41    } while (local_14 >> 1 != 0);
42    if ((((local_30 < 1) || (local_30 % 3 != 0)) || (local_30 != (local_30 / 9) * 9)) ||
43       ((float)local_14 != (float)local_30 / 27.00000000)) {
44      _printf("Checksum SHA-5119 Invalido!!!");
45      iVar2 = -1;
46    }
47    else {
48      local_5c[0] = 0x7b;
49      local_5c[1] = 0x66;
```

En este reto la gran dificultad radicaba en pasar todas las condiciones del binario a un script para que **z3** pudiera arrojarnos la flag.

Aqui el script final:

```
#! /usr/bin/python
from z3 import *

# tam flag == 27
tam_flag = 27
flag = [BitVec('val_%i'%i,16) for i in range(0, tam_flag)]

# Creating the solver
```

```
s = Solver()

################################

array_check = []
for _ in range(8): array_check.append(0)
array_check[0] = 0x7b
array_check[1] = 0x66
array_check[2] = 0x65
array_check[3] = 100
array_check[4] = 0x6b
array_check[5] = 99
array_check[6] = 0x61
array_check[7] = 0x68

idx_check_1 = 7
while(-1 < idx_check_1):
    s.add(flag[idx_check_1] == array_check[7 - idx_check_1])
    idx_check_1 -= 1

################################

s.add(flag[tam_flag - 1] == ord('}'))

################################

ini_interno = len("hackdef{")
# interno[5] == '-'
s.add(flag[ini_interno + 5] == ord('-'))
# interno[11] == '-'
s.add(flag[ini_interno + 11] == ord('-'))

################################

# interno[8] == '_'
s.add(flag[ini_interno + 8] == ord('_'))

################################
```

```
# interno[1] es digito
s.add(flag[ini_interno + 1] >= ord('0'))
s.add(flag[ini_interno + 1] <= ord('9'))

# interno[2] es digito
s.add(flag[ini_interno + 2] >= ord('0'))
s.add(flag[ini_interno + 2] <= ord('9'))

# interno[4] es digito
s.add(flag[ini_interno + 4] >= ord('0'))
s.add(flag[ini_interno + 4] <= ord('9'))

# interno[7] es digito
s.add(flag[ini_interno + 7] >= ord('0'))
s.add(flag[ini_interno + 7] <= ord('9'))

# interno[9] es digito
s.add(flag[ini_interno + 9] >= ord('0'))
s.add(flag[ini_interno + 9] <= ord('9'))

# interno[0xd] es digito
s.add(flag[ini_interno + 0xd] >= ord('0'))
s.add(flag[ini_interno + 0xd] <= ord('9'))

# interno[0x10] es digito
s.add(flag[ini_interno + 0x10] >= ord('0'))
s.add(flag[ini_interno + 0x10] <= ord('9'))

# interno[3] es minuscula
s.add(flag[ini_interno + 3] >= ord('a'))
s.add(flag[ini_interno + 3] <= ord('z'))

# interno[0xc] es minuscula
s.add(flag[ini_interno + 0xc] >= ord('a'))
s.add(flag[ini_interno + 0xc] <= ord('z'))

# interno[0xf] es minuscula
```

```
s.add(flag[ini_interno + 0xf] >= ord('a'))
s.add(flag[ini_interno + 0xf] <= ord('z'))

# interno[10] es minuscula
s.add(flag[ini_interno + 10] >= ord('a'))
s.add(flag[ini_interno + 10] <= ord('z'))

# interno[0] es mayuscula
s.add(flag[ini_interno + 0] >= ord('A'))
s.add(flag[ini_interno + 0] <= ord('Z'))

# interno[6] es mayuscula
s.add(flag[ini_interno + 6] >= ord('A'))
s.add(flag[ini_interno + 6] <= ord('Z'))

# interno[0xe] es mayuscula
s.add(flag[ini_interno + 0xe] >= ord('A'))
s.add(flag[ini_interno + 0xe] <= ord('Z'))

# interno[0x11] es mayuscula
s.add(flag[ini_interno + 0x11] >= ord('A'))
s.add(flag[ini_interno + 0x11] <= ord('Z'))

#################################

# final_check_1
s.add(flag[ini_interno + 4] * flag[ini_interno + 3] + (flag[ini_interno + 1] * fla
s.add(flag[ini_interno + 0] + flag[ini_interno + 1] + flag[ini_interno + 2] + flag
s.add((flag[ini_interno + 3] * flag[ini_interno + 2]) % flag[ini_interno + 4] == 0

#################################

# final_check_2
s.add((flag[ini_interno + 9] + flag[ini_interno + 7]) * (flag[ini_interno + 6] + f
s.add((flag[ini_interno + 10] - flag[ini_interno + 9]) * (flag[ini_interno + 6] +

#################################
```

```
# final_check_3
s.add((flag[ini_interno + 0xd] * flag[ini_interno + 0xc]) % flag[ini_interno + 0xe
s.add((flag[ini_interno + 0x10] * flag[ini_interno + 0xf]) % flag[ini_interno + 0x
s.add(flag[ini_interno + 0x11] * flag[ini_interno + 0x10] + flag[ini_interno + 0xc
s.add(flag[ini_interno + 0x11] + flag[ini_interno + 0xc] + flag[ini_interno + 0xd]

# checking constraings
if(s.check() == sat):
    m = s.model()
    cad = ""
    for i in range(tam_flag):
        cad += chr(m[flag[i]].as_long())
    print cad
else:
    print "Sorry ... :("
```

El unico detalle que tuvimos fue que cuando terminamos de pasar todas las condiciones y ejecutar el script nos arrojaba el mensaje de que no se había encontrado solución y perdimos un buen rato buscando que condición habíamos pasado mal. Pero el fallo estaba en que declaramos los caracteres de la flag como **BitVec** de **8 bits** y al parecer como habia ciertas condiciones que requerian multiplicaciones, eso provocaba que se desbordaran los valores y no encontrara la solución. Así que lo cambiamos a **BitVec** de **16 bits** y al ejecutarlo nos arrojó la flag.

```
$ ./script.py
hackdef{U51n6-Z3_1s-f4St3R}
$
```