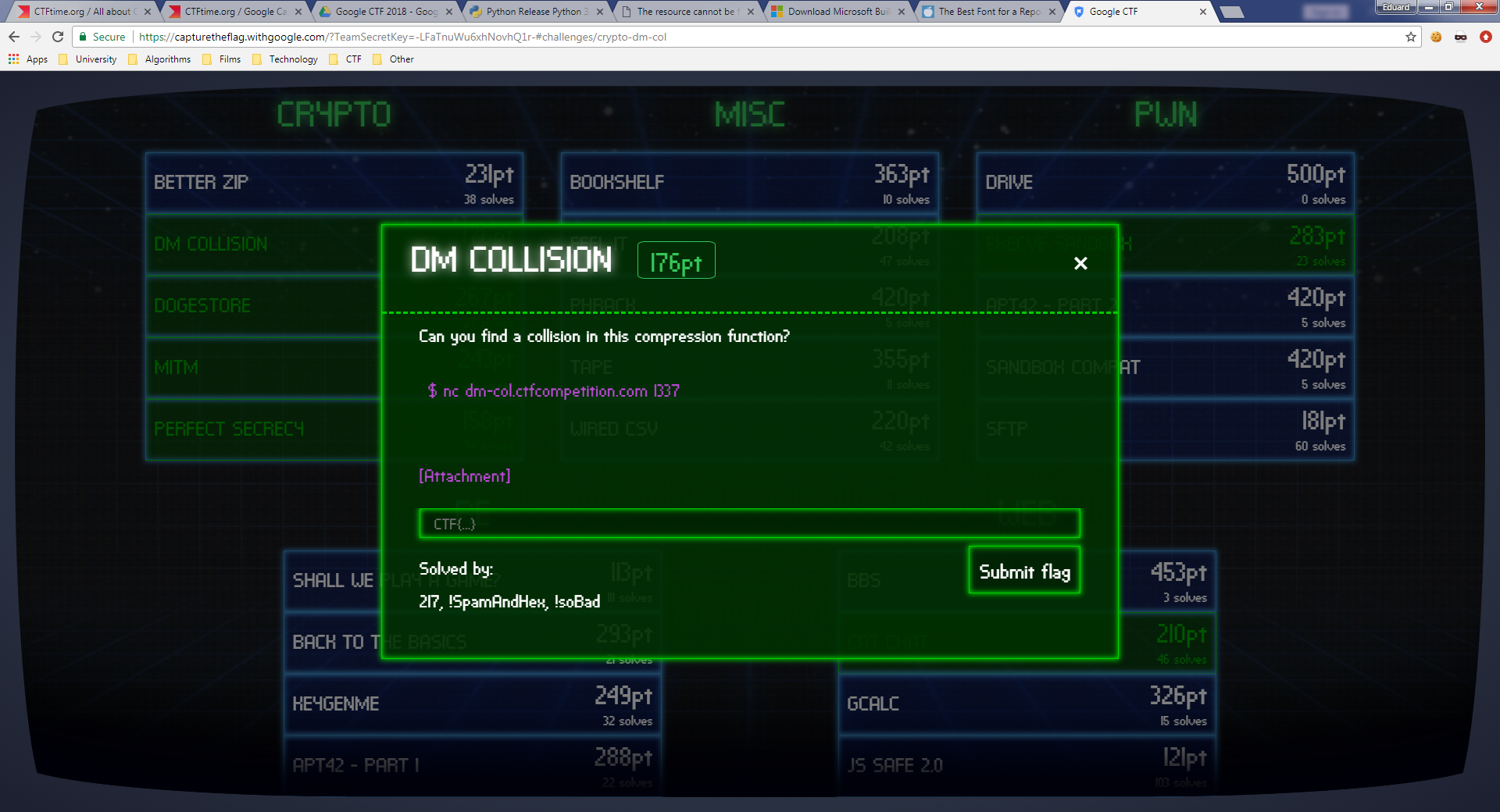
Google CTF 2018

DM Collision Writeup

shediyo @ PseudoRandom



From the challenge code I see that I have to find a collision and a preimage for 0 for the compression fuction.

The compression function is a Davis-Mayer compression from 16 to 8bytes through interpretation of the 16-byte block as a key of 8 bytes and plaintext of 8 bytes and the output block is , where is a modified cipher for which the exact implementation is also given. ( is the xor operation, is concetanation)

Finding a collision

To find a collision one needs to find plaintexts and keys so that:

In the case the goal is simplified:

That is, finding two different keys for which the DES' encryption gives the same output for the same plaintext .

Looking at the key scheduling algorithm:

*# Only 56 bits are used. A bit in each byte is reserved for parity checks.*

C = [key[PC1\_C[i] - 1] **for** i **in** range(28)]

D = [key[PC1\_D[i] - 1] **for** i **in** range(28)]

Since only 56 bits of the key are used, but the input key is 64 bits – we can swap one of the unused bits in any key and get a key for which we get the same result.

For example:

Finding a 0-preimage

To find a 0-preimage one needs to find a plaintext and a key so that:

DES is a Feistel-Network (FN) cipher, and the input of 64 bits is split after a permutation to 2 blocks of 32 bits, left and right - .

In round – for input , the output is , where is the round key for round .

Choosing in round the transition is .

The goal can now be changed for finding a 32-bit input and a 64-bit key for which for each round , that's because the FN will be doing no effect .

Looking again at the key scheduling algorithm, I see that the round keys are just rotations of 2 blocks of 28 bits:

**for** ri **in** range(16):

C = LeftShift(C, KS\_SHIFTS[ri])

D = LeftShift(D, KS\_SHIFTS[ri])

CD = Concat(C, D)

ki = [CD[PC2[i] - 1] **for** i **in** range(48)]

**yield** ki

Choosing C as or and D as or we get the same round key for each round. (notation - is a concatenation of zeroes)

So, for 4 special keys (after permutation) the goal is even simpler - finding for which .

Now I look at the cipher function:

*# Confusion step.*

res = Xor(Expand(inp), key)

sbox\_out = []

**for** si **in** range(48 // 6):

sbox\_inp = res[6 \* si:6 \* si + 6]

sbox = SBOXES[si]

row = (int(sbox\_inp[0]) << 1) + int(sbox\_inp[-1])

col = int(''.join([str(b) **for** b **in** sbox\_inp[1:5]]), 2)

bits = bin(sbox[row][col])[2:]

bits = '0' \* (4 - len(bits)) + bits

sbox\_out += [int(b) **for** b **in** bits]

*# Diffusion step.*

res = sbox\_out

res = [res[P[i] - 1] **for** i **in** range(32)]

**return** res

The cipher function in 3 steps:

1. The input is expanded from 32 bits to an expanded input of 48 bits via
2. The expanded input passes as eight 6-bit blocks through sboxes which are from 6 to 4 bits, outputting 32 bits (eight 4-bit blocks).
3. The output is bitwise-permuted.

If in step 2 I'll have a 0 output from each sbox, in step 3 the permutation will have no effect and I'll still have a 0 output from the cipher.

I saw that there are 4 preimages of 0 for each sbox, and so preimages of 0 in step 2 (as expanded input).

To succeed finding a cipher preimage I have a constraint from step 1 - the sboxes preimage must be an expansion of a 32-bit input, that is for some , otherwise I won't get a valid input for the cipher function.

My strategy of checking whether I have such an input is recursively finding the preimages for each sbox, and then passing the known bits as constraints to the following recursive solution of the next sbox, until I get a full legitimate input.

Implemented in the following way:

**def** rec\_preimage\_cipher(stage, first\_val, second\_val, val\_32, val\_1, key=(0,0)):

sbox\_inp = [-1, -1, -1, -1, -1, -1]

sbox\_inp[0], sbox\_inp[1] = first\_val, second\_val

**for** i **in** range(2 \*\* 4):

sbox\_inp[2], sbox\_inp[3] = (i % 2), ((i // 2) % 2)

sbox\_inp[4] = val\_32 **if** stage == 7 **else** ((i // 4) % 2)

sbox\_inp[5] = val\_1 **if** stage == 7 **else** ((i // 8) % 2)

keyed\_sbox\_inp = [0] \* 6

**for** j **in** range(6):

keyed\_sbox\_inp[j] = sbox\_inp[j] ^ key[stage // 4]

sbox = SBOXES[stage]

row = (int(keyed\_sbox\_inp[0]) << 1) + int(keyed\_sbox\_inp[-1])

col = int(''.join([str(b) **for** b **in** keyed\_sbox\_inp[1:5]]), 2)

**if** sbox[row][col] == 0:

**if** stage == 7:

**return** sbox\_inp

mid\_res = rec\_preimage\_cipher(stage + 1, sbox\_inp[4], sbox\_inp[5], val\_32, val\_1, key)

**if** mid\_res **is** **not** None:

**return** sbox\_inp + mid\_res

**return** None

**def** preimage\_cipher():

chosen\_input, final\_key = None, None

**for** key **in** [(0,0), (0,1), (1,0), (1,1)]:

**for** t **in** [(0,0), (0,1), (1,0), (1,1)]:

res = rec\_preimage\_cipher(0, t[0], t[1], t[0], t[1], key)

**if** res **is** **not** None:

chosen\_input = res

final\_key = key

**break**

I got one good preimage input and key:

Both of them are after permutations, inversing the permutations I got:

After sending all the results to the server I got the flag:

CTF{7h3r35 4 f1r3 574r71n6 1n my h34r7 r34ch1n6 4 f3v3r p17ch 4nd

175 br1n61n6 m3 0u7 7h3 d4rk}