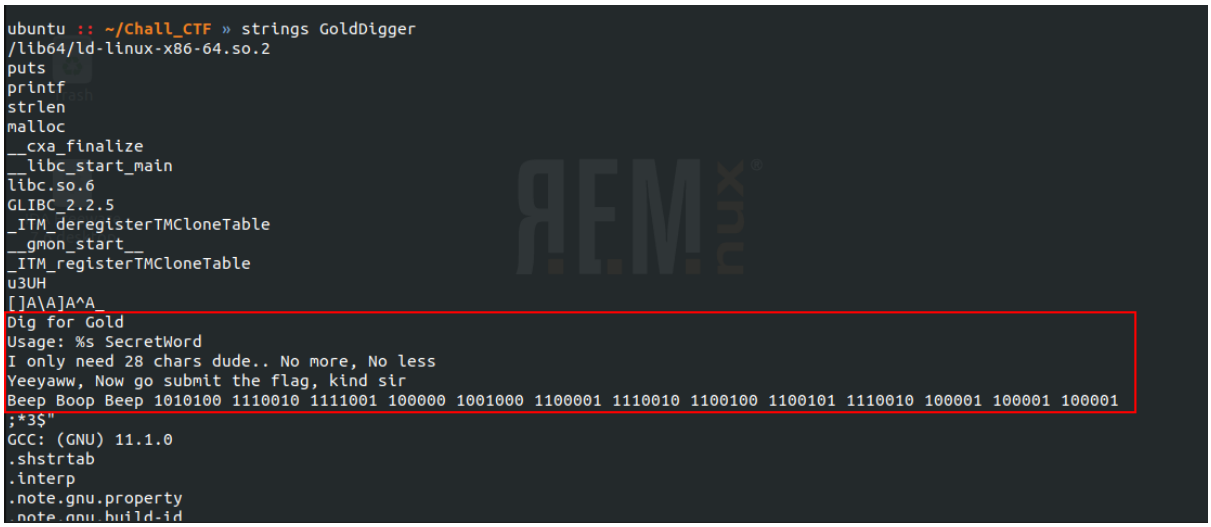
GoldDigger

So as always first let's check what file type it is.

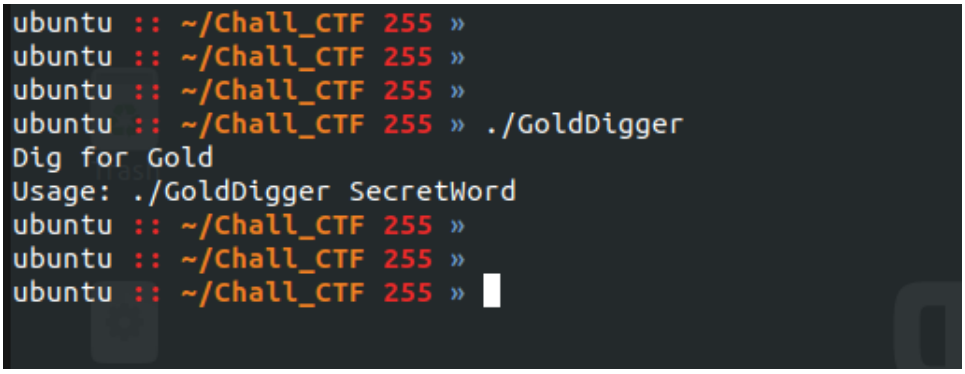
We can see this is a ELF 64-bit binary, and it is stripped.

Before running the binary let's see if there is any juicy hard-coded strings.

There is no juicy information. But we see some readable strings.

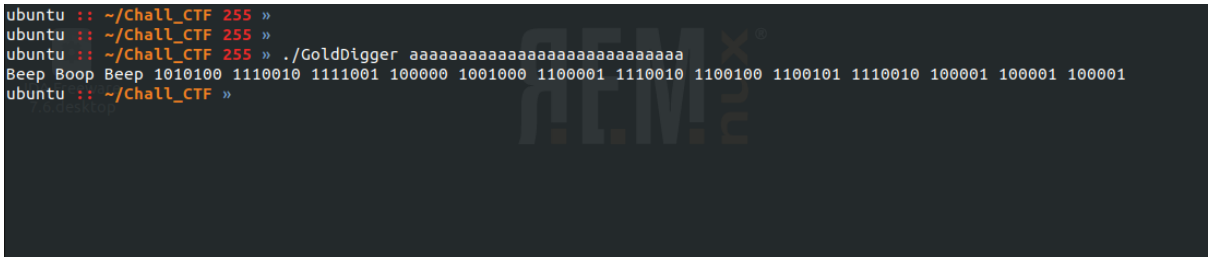


When executing the binary we see that it asks for a secret.

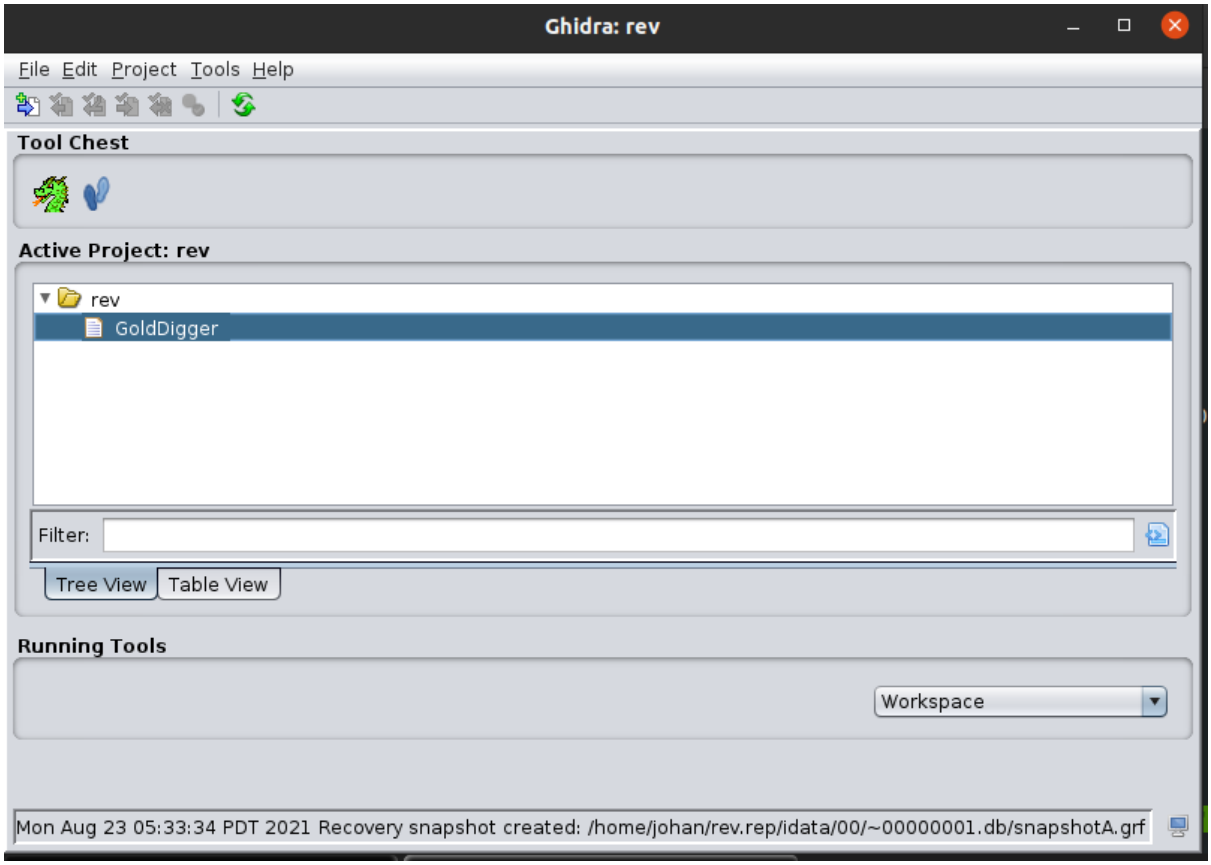


And we also see that the secret is 28 charters long.

When giving the wrong secret it just gives a fail message. So let's decompile and take a look at the inside of the binary.

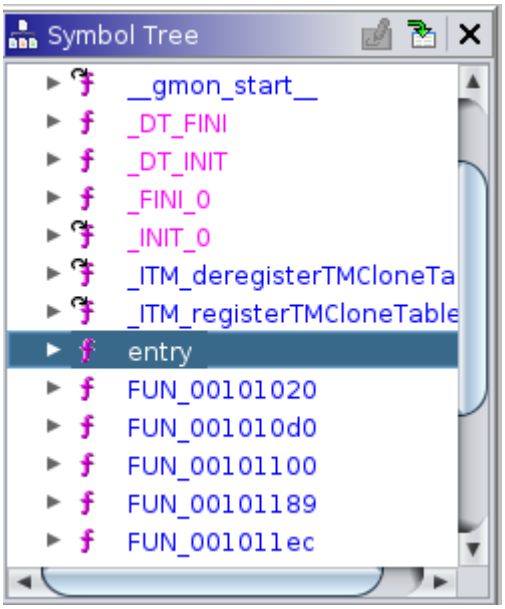


Let's open the binary in Ghidra. And open the code browser.

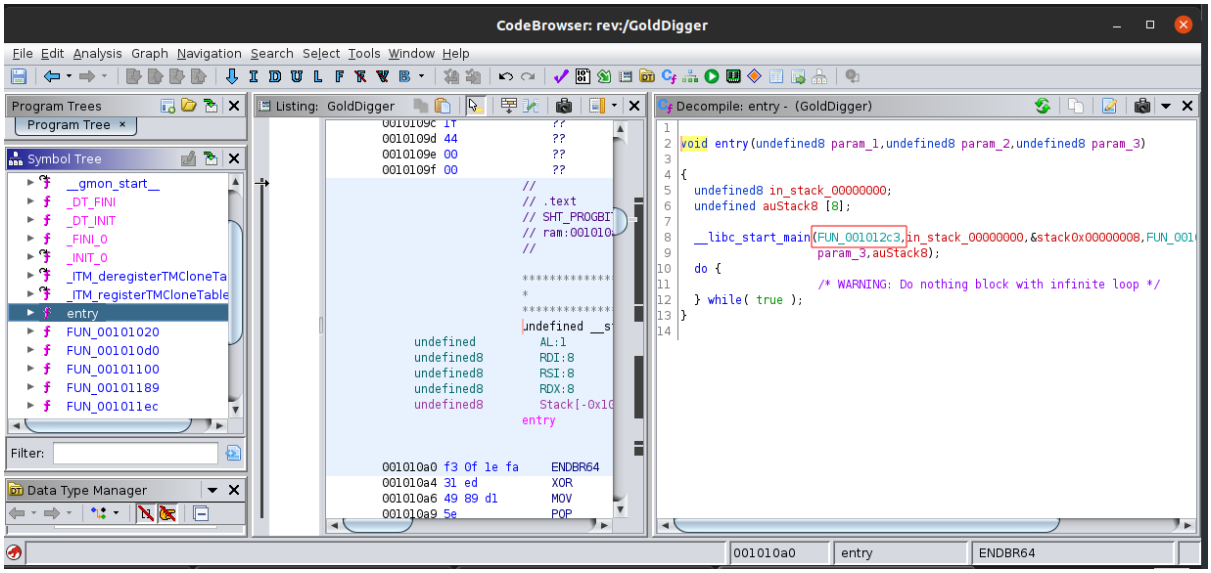


We some functions in the binary. So let's find the main fucntion.

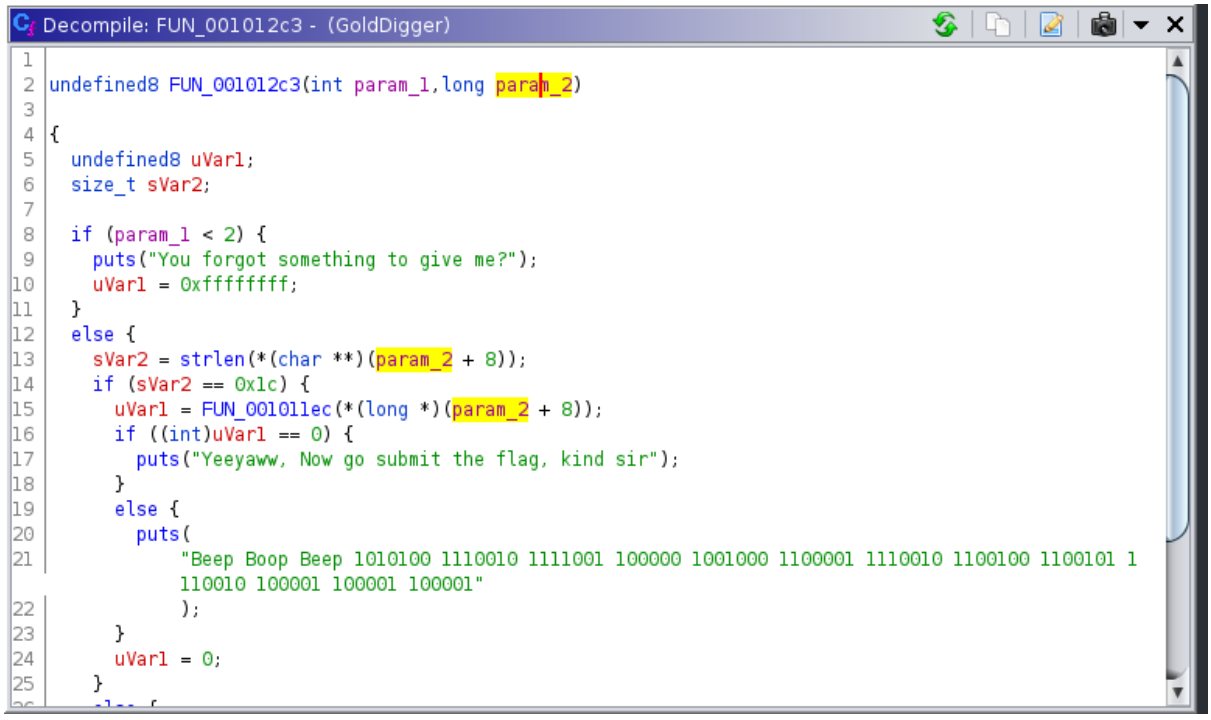
Usually the entry functions will call the next function it wants to load(main). So let's see what's in entry.



We see `__libc_start_main` takes one function(`FUN_001012c3()`) as it's argument . And that is our main function.



The main function takes one argument which is our input. When the length of our input is `0x1c(28)` characters long, the input goes into another function (`FUN_001011e()`). The return state of that function is saved. And based on the return state if it is 0 (no errors) then it'll give us the success message.



The function `FUN_001011e()` calls another function called `FUN_00101189()` inside it.

Decompile: FUN_001011ec - (GoldDigger)

```
1
2  undefined8 FUN_001011ec(long param_1)
3
4  {
5    void *pvVar1;
6    void *pvVar2;
7    int local_20;
8
9    pvVar1 = malloc(0x10);
10   pvVar2 = FUN_00101189();
11   for (local_20 = 0; local_20 < 0x1c; local_20 = local_20 + 1) {
12     *(int *)((long)pvVar1 + (long)local_20 * 4) =
13          *(char *)(param_1 + *(int *)((long)pvVar2 + (long)local_20 * 4)) + 4;
14   }
15   local_20 = 0;
16   while( true ) {
17     if (0x1b < local_20) {
18       return 0;
19     }
20     if (*(int *)((long)pvVar1 + (long)local_20 * 4) != *(int *)(&DAT_001020a0 + (long)local_20 * 4))
21     break;
22     local_20 = local_20 + 1;
23   }
24   return 0xffffffff;
25 }
26
```

This function isn't doing much.

Decompile: FUN_00101189 - (GoldDigger)

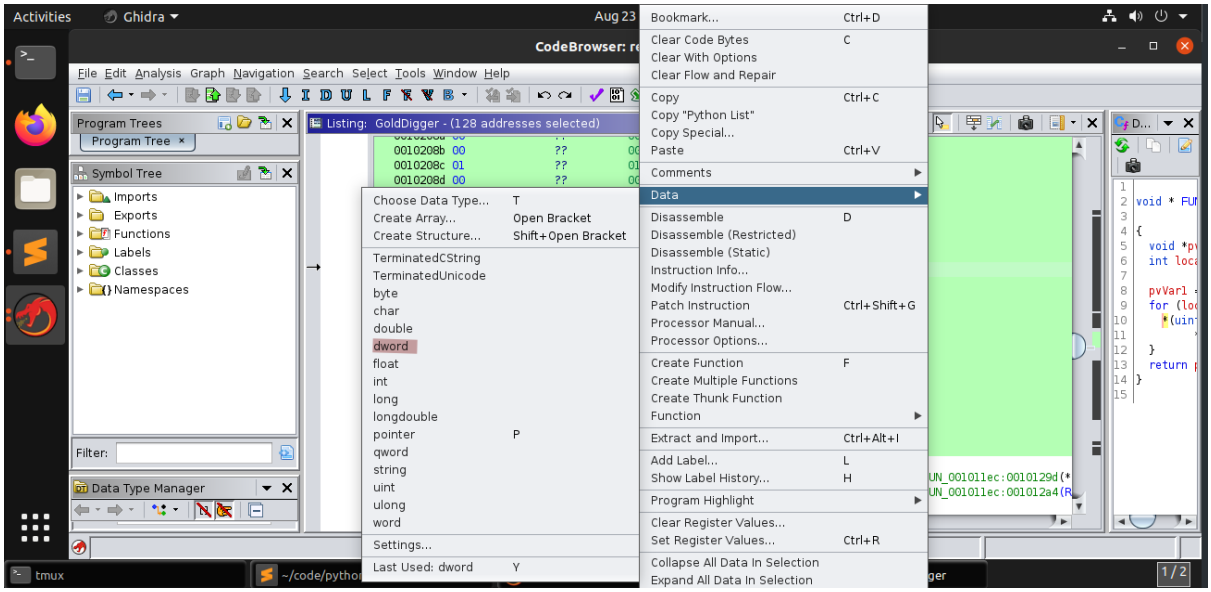
```
1
2  void * FUN_00101189(void)
3
4  {
5    void *pvVar1;
6    int local_14;
7
8    pvVar1 = malloc(0x1c0);
9    for (local_14 = 0; local_14 < 0x1c; local_14 = local_14 + 1) {
10     *(uint *)((long)pvVar1 + (long)local_14 * 4) =
11          *(uint *)(&DAT_00102020 + (long)local_14 * 4) ^ 0x12;
12   }
13   return pvVar1;
14 }
15
```

Code explanation

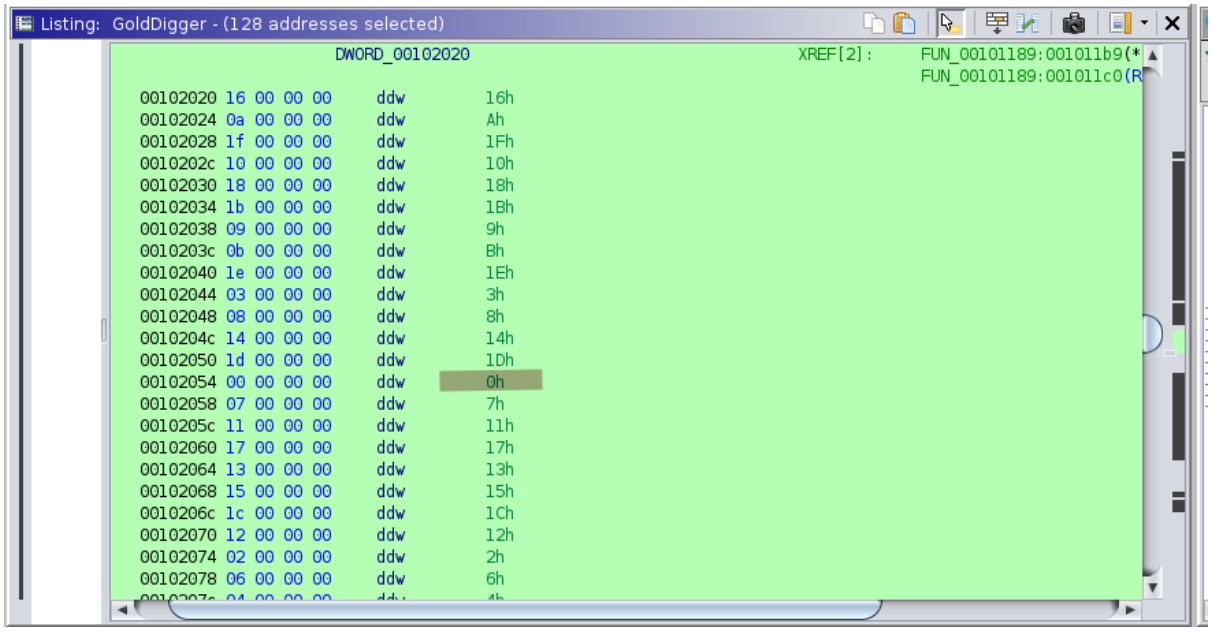
```
void * FUN_00101189(void)

{
  void *pvVar1;
  int local_14;
```

```
  pvVar1 = malloc(0x1c0); //allocating some memory block

  for (local_14 = 0; local_14 < 0x1c; local_14 = local_14 + 1) { //doing a 0x1c long l
oop
//getting the data from the location DAT_00102020 and xors each object with 0x12.
//and saves the result on the allocated memory pvVar1.
    *(uint *)((long)pvVar1 + (long)local_14 * 4) = *(uint *)(&DAT_00102020 + (long)loc
al_14 * 4) ^ 0x12;
  }
  return pvVar1; // returns the result.
}
```

Inside the location `&DAT_00102020` we find some data which looks like array. Let's convert the data type into Dword. The reason for this is, usually we'll copy the bytes and try to filter the `\x00` padding. But if there is any zeros value in the array it'll make us miss that zero value while filtering out. So let's convert the data type and see if there is any zero values.

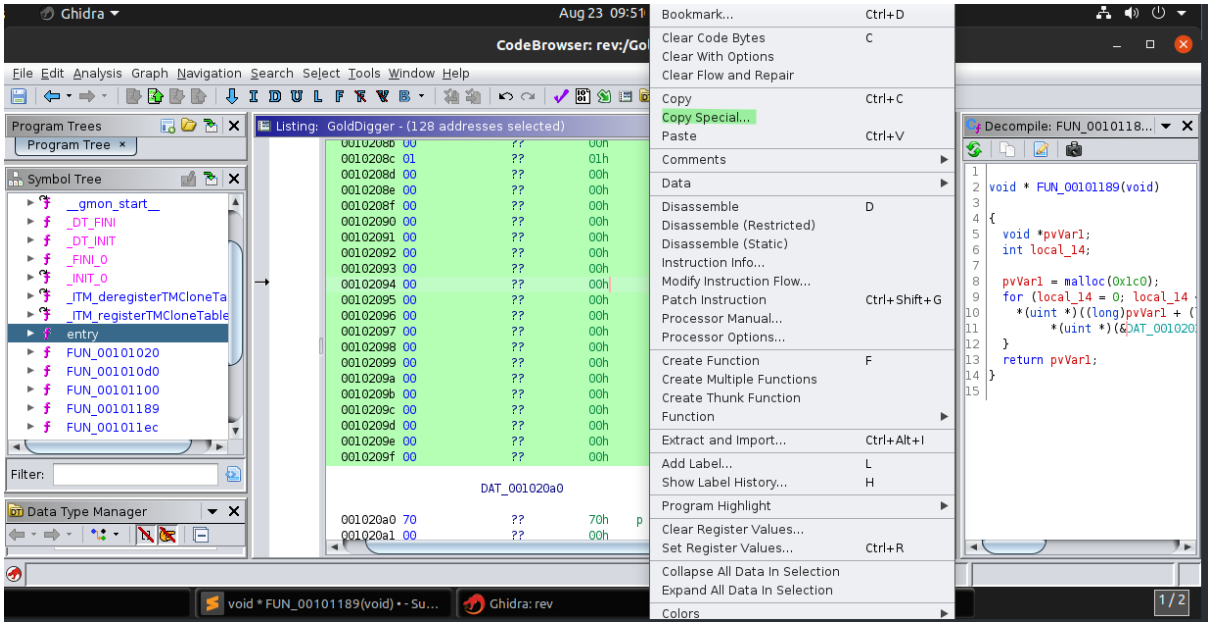


As expected there is a zero value. This means we should add a zero value in the array after filtering the padding.



let's copy these data and see if we can make any sense from it.

We can copy this piece of data as python list by selecting the whole location and right clicking on the location and choosing Copy Special. Choose python list from the options.

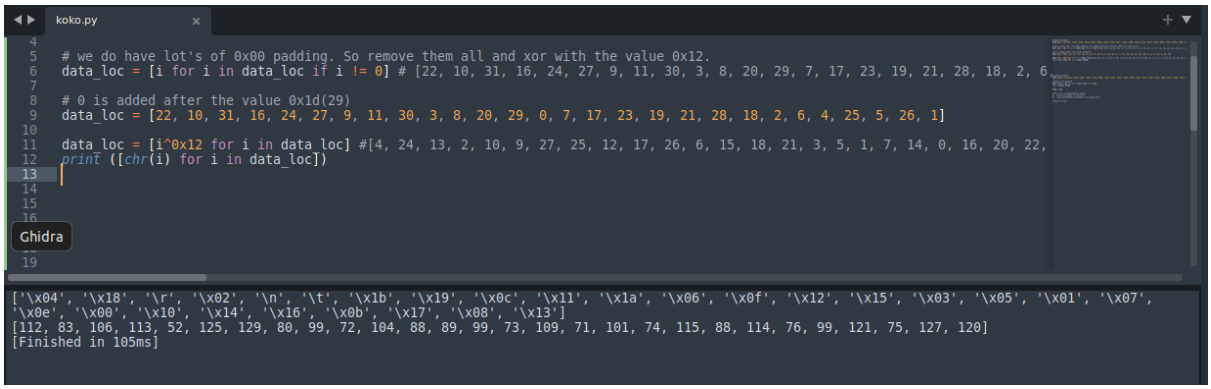


Now let's recreate this scene in python.

```
# &DAT_00102020
data_loc = [ 0x16, 0x00, 0x00, 0x00, 0x0a, 0x00, 0x00, 0x00, 0x1f, 0x00, 0x00, 0x00, 0
x10, 0x00, 0x00, 0x00, 0x18, 0x00, 0x00, 0x00, 0x1b, 0x00, 0x00, 0x00, 0x09, 0x00, 0x0
0, 0x00, 0x0b, 0x00, 0x00, 0x00, 0x1e, 0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x08,
```

```
0x00, 0x00, 0x00, 0x14, 0x00, 0x00, 0x00, 0x1d, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x
00, 0x07, 0x00, 0x00, 0x00, 0x11, 0x00, 0x00, 0x00, 0x17, 0x00, 0x00, 0x00, 0x13, 0x0
0, 0x00, 0x00, 0x15, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00, 0x12, 0x00, 0x00, 0x00,
0x02, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x19, 0x00, 0x
00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x1a, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x0
0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00 ]

# we do have lot's of 0x00 padding. So remove them all and xor with the value 0x12.
data_loc = [i for i in data_loc if i != 0] # [22, 10, 31, 16, 24, 27, 9, 11, 30, 3, 8,
20, 29, 7, 17, 23, 19, 21, 28, 18, 2, 6, 4, 25, 5, 26, 1]

# 0 is added after the value 0x1d(29)
data_loc = [22, 10, 31, 16, 24, 27, 9, 11, 30, 3, 8, 20, 29, 0, 7, 17, 23, 19, 21, 28,
18, 2, 6, 4, 25, 5, 26, 1]

data_loc = [i^0x12 for i in data_loc] #[4, 24, 13, 2, 10, 9, 27, 25, 12, 17, 26, 6, 1
5, 18, 21, 3, 5, 1, 7, 14, 0, 16, 20, 22, 11, 23, 8, 19]
```

We got a result whose length is 28 chars long just as the secret we expected. But it isn't giving any meaningful data.



Let's go back to the main function.

The Xor'ed result is getting stored in `pvVar2` variable here. And there is another loop.

This Loop gets our input and increase the ascii value of every character by 4 and save it in `pvVar1` based on the location it gets from `FUN_001011c8()`. To put it simple, It just get our input increase the ascii value of each char by 4 and shuffles the array.

if the first value `FUN_001011c8(`) returns is 12, then the logic will look something like this

> pvVar1[0] = param_1[12] (input) + 4 (increase the ascii value of that character)

Decompile: FUN_001011c8 - (GoldDigger)

```
1
2  undefined8 FUN_001011c8(long param_1)
3
4  {
5    void *pvVar1;
6    void *pvVar2;
7    int local_20;
8
9    pvVar1 = malloc(0x10);
10   pvVar2 = FUN_00101169();
11   for (local_20 = 0; local_20 < 0x1c; local_20 = local_20 + 1) {
12     *(int *)((long)pvVar1 + (long)local_20 * 4) =
13          *(char *)(param_1 + *(int *)((long)pvVar2 + (long)local_20 * 4)) + 4;
14   }
15   local_20 = 0;
16   while( true ) {
17     if (0x1b < local_20) {
18       return 0;
19     }
20     if (*(int *)((long)pvVar1 + (long)local_20 * 4) != *(int *)(&DAT_001020a0 + (long)local_20 * 4))
21     break;
22     local_20 = local_20 + 1;
23   }
24   return 0xffffffff;
25 }
26
```

Just to be clear:

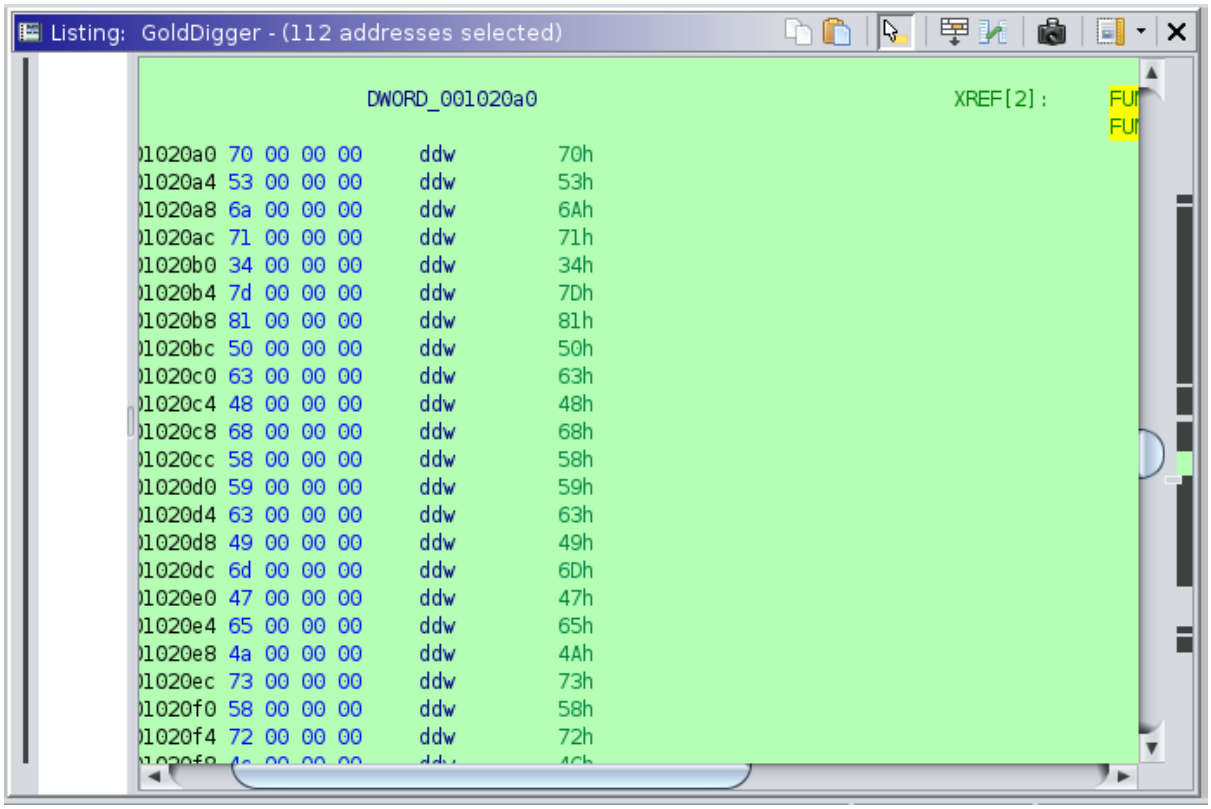
`*(int *)((long)pvVar1 + (long)local_20 * 4) = *(char *)(param_1 + *(int *)((long)pvVar2 + (long)local_20 * 4)) + 4;`

> pvVar1 is the location of array. And by adding local_20 it gradually jumps to next location. local_20 is multiplied by 4 here cuz the data in the memory will be padded with `0x00` so that it will represent a valid address. So 0xa will look like `0x0000000a` something like this in the memory.

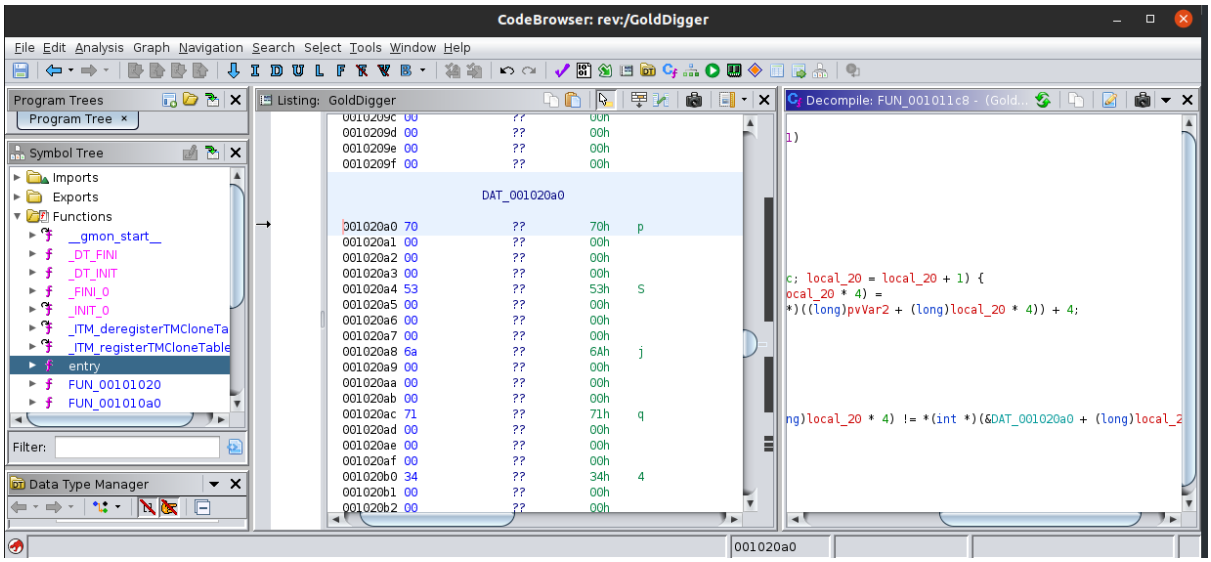
Lastly it compares the resulted data with another location (`&DAT_001020a0`) of arrays. If they're same then we got our secret.

Let's convert the data type and check for zero values, then get the values from that location.

As seen below there is no zero values.

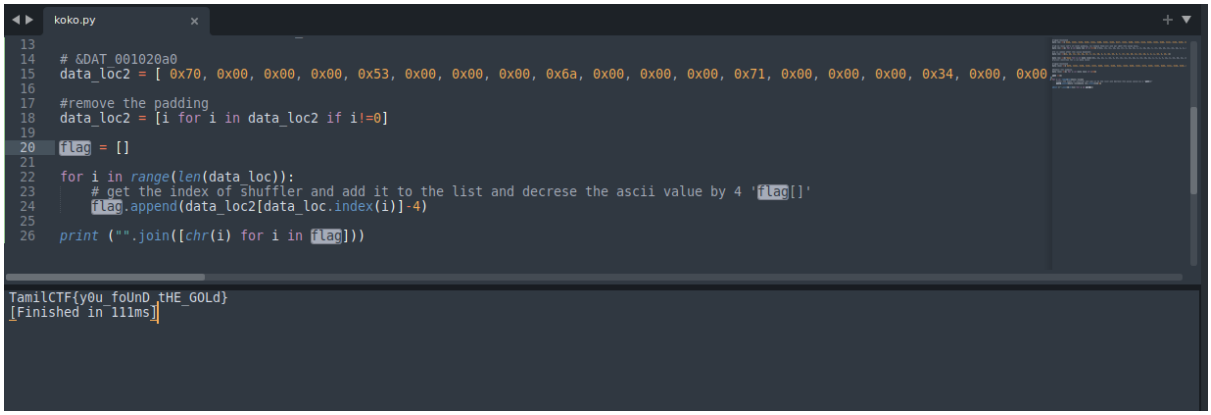


Again we can copy this data as python list and manually. We already have the shuffle order that we got from `FUN_00101169()`. Now we have everything we need to reverse this function.



So with the following script we can get the flag.

```
# &DAT_00102020
data_loc = [ 0x16, 0x00, 0x00, 0x00, 0x0a, 0x00, 0x00, 0x00, 0x1f, 0x00, 0x00, 0x00, 0
x10, 0x00, 0x00, 0x00, 0x18, 0x00, 0x00, 0x00, 0x1b, 0x00, 0x00, 0x00, 0x09, 0x00, 0x0
0, 0x00, 0x0b, 0x00, 0x00, 0x00, 0x1e, 0x00, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00, 0x08,
0x00, 0x00, 0x00, 0x14, 0x00, 0x00, 0x00, 0x1d, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x
00, 0x07, 0x00, 0x00, 0x00, 0x11, 0x00, 0x00, 0x00, 0x17, 0x00, 0x00, 0x00, 0x13, 0x0
0, 0x00, 0x00, 0x15, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00, 0x12, 0x00, 0x00, 0x00,
0x02, 0x00, 0x00, 0x00, 0x06, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x19, 0x00, 0x
00, 0x00, 0x05, 0x00, 0x00, 0x00, 0x1a, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x0
0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
0x00 ]

# we do have lot's of 0x00 padding. So remove them all and xor with the value 0x12.
data_loc = [i for i in data_loc if i != 0] # [22, 10, 31, 16, 24, 27, 9, 11, 30, 3, 8,
20, 29, 7, 17, 23, 19, 21, 28, 18, 2, 6, 4, 25, 5, 26, 1]

# 0 is added after the value 0x1d(29)
data_loc = [22, 10, 31, 16, 24, 27, 9, 11, 30, 3, 8, 20, 29, 0, 7, 17, 23, 19, 21, 28,
18, 2, 6, 4, 25, 5, 26, 1]

data_loc = [i^0x12 for i in data_loc] #[4, 24, 13, 2, 10, 9, 27, 25, 12, 17, 26, 6, 1
5, 18, 21, 3, 5, 1, 7, 14, 0, 16, 20, 22, 11, 23, 8, 19]
# print ([chr(i) for i in data_loc])

# &DAT_001020a0
data_loc2 = [ 0x70, 0x00, 0x00, 0x00, 0x53, 0x00, 0x00, 0x00, 0x6a, 0x00, 0x00, 0x00,
 0x71, 0x00, 0x00, 0x00, 0x34, 0x00, 0x00, 0x00, 0x7d, 0x00, 0x00, 0x00, 0x81, 0x00, 0
x00, 0x00, 0x50, 0x00, 0x00, 0x00, 0x63, 0x00, 0x00, 0x00, 0x48, 0x00, 0x00, 0x00, 0x6
8, 0x00, 0x00, 0x00, 0x58, 0x00, 0x00, 0x00, 0x59, 0x00, 0x00, 0x00, 0x63, 0x00, 0x00,
0x00, 0x49, 0x00, 0x00, 0x00, 0x6d, 0x00, 0x00, 0x00, 0x47, 0x00, 0x00, 0x00, 0x65, 0x
00, 0x00, 0x00, 0x4a, 0x00, 0x00, 0x00, 0x73, 0x00, 0x00, 0x00, 0x58, 0x00, 0x00, 0x0
0, 0x72, 0x00, 0x00, 0x00, 0x4c, 0x00, 0x00, 0x00, 0x63, 0x00, 0x00, 0x00, 0x79, 0x00,
0x00, 0x00, 0x4b, 0x00, 0x00, 0x00, 0x7f, 0x00, 0x00, 0x00, 0x78, 0x00, 0x00, 0x00 ]

#remove the padding
data_loc2 = [i for i in data_loc2 if i!=0]
print (data_loc2)

flag = []

for i in range(len(data_loc)):
  # get the index of shuffler and add it to the list and decrese the ascii value by 4
 'flag[]'
  flag.append(data_loc2[data_loc.index(i)]-4)

print ("".join([chr(i) for i in flag]))
```



We can verify the flag with the binary