We are given an amd64 ELF executable, which once executed, copies a slightly modified version of itself under a name doll-N, where N is a current executable number +1.

```
root@cdca89554323:~# ls
doll
root@cdca89554323:~# ./doll
Unwrapping the 1-th doll
root@cdca89554323:~# ls
doll  doll-1
root@cdca89554323:~# ./doll-1
Unwrapping the 2-th doll
root@cdca89554323:~# ./doll-2
Unwrapping the 3-th doll
root@cdca89554323:~# ./doll-3
Unwrapping the 4-th doll
root@cdca89554323:~# ./doll-4
Unwrapping the 5-th doll
root@cdca89554323:~# ls
doll  doll-1  doll-2  doll-3  doll-4  doll-5
root@cdca89554323:~#
```

Let’s try and decompile the executable with HexRays (functions and variables were renamed after analysis):

```c
__int64 __fastcall main(int a1, char **a2, char **a3)
{
  __int64 v4; // rax
  __int64 rnd; // [rsp+8h] [rbp-38h]
  __int64 new_data[4]; // [rsp+10h] [rbp-30h] BYREF
  unsigned __int64 canary; // [rsp+38h] [rbp-8h]

  canary = __readfsqword(0x28u);
  if ( n_doll == 8338671502332819874LL )
  {
    print_flag();
  }
  else
  {
    printf("Unwrapping the %llu-th doll\n", n_doll + 1);
    new_data[0] = some_const;
    new_data[1] = first_qword;
    new_data[3] = n_doll;
    rnd = random_int64();
    new_data[2] = sub_1373(second_qword, rnd);
    v4 = sub_1373(new_data[1], 11LL);
    new_data[1] = sub_1373(v4, rnd);
    ++new_data[3];
    generate_doll(new_data);
  }
  return 0LL;
}
```

With data defined as follows:

```
                align 20h
some_const      dq 5999FFAE750B4AF1h    ; DATA XREF: main+62↑r
first_qword     dq 1                    ; DATA XREF: print_flag+2D↑r
                                        ; main+69↑r
second_qword    dq 1                    ; DATA XREF: print_flag+1B↑r
                                        ; main+78↑r ...
n_doll          dq 0                    ; DATA XREF: main+1B↑r
                                        ; main:loc_1921↑r ...
_data           ends
```

The first idea would be of course to replace n_doll with 8338671502332819874 and trigger print_flag. But after looking at print_flag function closely, we can see that it uses first_qword and second_qword variables to calculate the flag. So the flag is actually contains in those two variables, and just changing the doll number won't help.

```
unsigned __int64 print_flag()
{
  __int64 v0; // rax
  __int64 v2; // [rsp+10h] [rbp-20h]
  __int64 v3; // [rsp+1Fh] [rbp-11h] BYREF
  char v4; // [rsp+27h] [rbp-9h]
  unsigned __int64 canary; // [rsp+28h] [rbp-8h]

  canary = __readfsqword(0x28u);
  v0 = sub_1442(second_qword);
  v2 = sub_1373(first_qword, v0);
  v3 = sub_1373(0x888BE665BFB73F2LL, v2);
  v4 = 0;
  puts("Congrats! You found the flag:");
  printf("sdctf{%s_%llu}\n", (const char *)&v3, v2);
  return canary - __readfsqword(0x28u);
}
```

We can see that the flag is calculated as a call to sub_1373 function with a 64-bit constant parameter and some unknown value v2, which is calculated from first_qword and second_qword, which are generated on each new doll unwrap.

The decompiled function sub_1373 does not give a clear idea of what the function does (at least to me lol). But of course it's obvious that the function performs some arithmetical operations that transform two 64-bit parameters into a 64-bit return value.

```
__int64 __fastcall sub_1373(__int64 a1, __int64 a2)
{
  __int64 v2; // rdx
  __int64 v3; // rdx
  int i; // [rsp+14h] [rbp-Ch]
  __int64 ret; // [rsp+18h] [rbp-8h]

  ret = 0LL;
  for ( i = 0; i <= 63; ++i )
  {
    v2 = (0x23B3 * (unsigned __int128)(unsigned __int64)(2 * ret)) >> 64;
    ret = 2 * ret - 0x7FFFFFFFFFFFEE27LL * ((v2 + ((unsigned __int64)(2 * ret - v2) >> 1)) >> 62);
    if ( a2 < 0 )
    {
      v3 = (0x23B3 * (unsigned __int128)(unsigned __int64)(ret + a1)) >> 64;
      ret = ret + a1 - 0x7FFFFFFFFFFFEE27LL * ((v3 + ((unsigned __int64)(ret + a1 - v3) >> 1)) >> 62);
    }
    a2 *= 2LL;
  }
  return ret;
}
```

What's interesting is, that on each new doll generation, new data uses a random 64-bit generated value to calculate and replace first_qword and second_qword (starting values for those are 1 and 1 as you can see from .data).

To simplify, the doll data is generated like that:

```
    uint64_t d1 = 1;
    uint64_t d2 = 1;
    for (uint64_t i = 0; i < 8338671502332819874; i++) {
```

```
        uint64_t r = rand();
        d2 = sub_1373(d2, r);
        d1 = sub_1373(d1, 11LL);
        d1 = sub_1373(d1, r);
    }
    uint64_t v0 = sub_1442(d2);
    uint64_t v2 = sub_1373(d1, v0);
    uint64_t v3 = sub_1373(0x888BE665BFB73F2LL, v2);
    puts("Congrats! You found the flag:");
    printf("sdctf{%s_%llu}\n", (const char *)&v3, v2);
```

It looks easy, but of course iterating through that loop will take forever.

We know that the flag is a constant value, which means random value r does not affect it. To simplify the calculation, let's assume that the random value is always 1.

Empirically, we have found out that sub_1373(n, 1) always returns n and sub_1442(1) returns 1. That means we can discard sub_1442 completely and also get rid of d2 (aka second_qword). The simplified code will look like this:

```
    uint64_t d1 = 1;
    for (uint64_t i = 0; i < 8338671502332819874; i++) {
        d1 = sub_1373(d1, 11LL);
    }
    uint64_t v3 = sub_1373(0x888BE665BFB73F2LL, d1);
    puts("Congrats! You found the flag:");
    printf("sdctf{%s_%llu}\n", (const char *)&v3, d1);
```

So all we need to do is to find d1 after 8338671502332819874 iterations, and we have the flag.

But, it's still unclear what sub_1373 function does. Let's print out first few values of the loop:

1: 11 (b)
2: 121 (79)
3: 1331 (533)
4: 14641 (3931)
5: 161051 (2751b)
6: 1771561 (1b0829)
7: 19487171 (12959c3)
8: 214358881 (cc6db61)
9: 2357947691 (8c8b6d2b)
10: 25937424601 (609fdb0d9)
11: 285311670611 (426de69953)
12: 3138428376721 (2dab8e89691)
13: 34522712143931 (1f65f1fe783b)
14: 379749833583241 (1596165ef2a89)
15: 4177248169415651 (ed72f6146d3e3)
16: 45949729863572161 (a33f092e0b1ac1)
17: 505447028499293771 (703b564fa7a264b)
18: 5559917313492231481 (4d28cb56c33fa539)
19: 5818858227285918857 (50c0bcba63bc8489)
20: 8667208279016479993 (78481c02491a1cf9)
21: 3105570700633567533 (2b193419241ff12d)
22: 6491161596404929146 (5a153d148d5f927a)
23: 6839173302470821933 (5ee99fe2131bc82d)

This looks like exponentiation of the initial parameter 11, but at some point it 'breaks', because we only have 64 bits. Maybe, it's a modular exponentiation with that weird constant from the function? Let's check:

```
sage: pow(11, 19, 0x7FFFFFFFFFFFEE27)
5818858227285918857
sage: pow(11, 20, 0x7FFFFFFFFFFFEE27)
8667208279016479993
sage: pow(11, 21, 0x7FFFFFFFFFFFEE27)
3105570700633567533
sage: pow(11, 22, 0x7FFFFFFFFFFFEE27)
6491161596404929146
sage: pow(11, 23, 0x7FFFFFFFFFFFEE27)
6839173302470821933
```

Yep, it is.

Which means we can easily get our d1:

```
sage: pow(11, 8338671502332819874, 0x7FFFFFFFFFFFEE27)
3865704192625469676
```

It appears that the function behaving like modular exponentiation only works for smaller numbers, so to get the actual flag, we just call the original function from C:

```
    uint64_t d1 = 3865704192625469676;
    uint64_t v3 = sub_1373(0x888BE665BFB73F2LL, d1);
    puts("Congrats! You found the flag:");
    printf("sdctf{%s_%llu}\n", (const char *)&v3, d1);
```

Let’s compile and run it and get the flag:

```
Congrats! You found the flag:
sdctf{sQU&Mu1t_3865704192625469676}
```