Reverse Solution

The binary is an ELF executable

```
file ctf
ctf: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=5e18d24f35571ae92c74469979823a6ba7272bf8, for GNU/Linux 3.2.0, not stripped
```

We take a look at the functions list in GDB

```
(gdb) info functions
All defined functions:

Non-debugging symbols:
0x0000000000001000  _init
0x0000000000001030  puts@plt
0x0000000000001040  strlen@plt
0x0000000000001050  printf@plt
0x0000000000001060  strcmp@plt
0x0000000000001070  __isoc99_scanf@plt
0x0000000000001080  __cxa_finalize@plt
0x0000000000001090  _start
0x00000000000010c0  deregister_tm_clones
0x00000000000010f0  register_tm_clones
0x0000000000001130  __do_global_dtors_aux
0x0000000000001170  frame_dummy
0x0000000000001179  deobfuscate
0x00000000000011c3  main
0x0000000000001274  _fini
```

To make `gdb` display the disassembled instruction automatically after each `stepi`, we can use:

`set disassemble-next-line on`

First of all we set a breakpoint in the `main` function

```
Breakpoint 1, 0x00005555555551c7 in main ()
=> 0x00005555555551c7 <main+4>: 48 83 ec 60            sub    $0x60,%rsp
(gdb) disas main
Dump of assembler code for function main:
   0x00005555555551c3 <+0>:     push   %rbp
   0x00005555555551c4 <+1>:     mov    %rsp,%rbp
=> 0x00005555555551c7 <+4>:     sub    $0x60,%rsp
   0x00005555555551cb <+8>:     movabs $0x1c0e193f313b3429,%rax
   0x00005555555551d5 <+18>:    movabs $0x3f3f6f6c6b636d21,%rdx
   0x00005555555551df <+28>:    mov    %rax,-0x20(%rbp)
   0x00005555555551e3 <+32>:    mov    %rdx,-0x18(%rbp)
   0x00005555555551e7 <+36>:    movl   $0x273f62,-0x10(%rbp)
   0x00005555555551ee <+43>:    lea    -0x20(%rbp),%rax
   0x00005555555551f2 <+47>:    mov    %rax,%rdi
   0x00005555555551f5 <+50>:    call   0x555555555040 <strlen@plt>
   0x00005555555551fa <+55>:    mov    %eax,%edx
   0x00005555555551fc <+57>:    lea    -0x20(%rbp),%rax
   0x0000555555555200 <+61>:    mov    %edx,%esi
   0x0000555555555202 <+63>:    mov    %rax,%rdi
   0x0000555555555205 <+66>:    call   0x555555555179 <deobfuscate>
   0x000055555555520a <+71>:    lea    0xdf3(%rip),%rax        # 0x555555556004
   0x0000555555555211 <+78>:    mov    %rax,%rdi
   0x0000555555555214 <+81>:    mov    $0x0,%eax
   0x0000555555555219 <+86>:    call   0x555555555050 <printf@plt>
   0x000055555555521e <+91>:    lea    -0x60(%rbp),%rax
   0x0000555555555222 <+95>:    mov    %rax,%rsi
   0x0000555555555225 <+98>:    lea    0xdf1(%rip),%rax        # 0x55555555601d
   0x000055555555522c <+105>:   mov    %rax,%rdi
   0x000055555555522f <+108>:   mov    $0x0,%eax
   0x0000555555555234 <+113>:   call   0x555555555070 <__isoc99_scanf@plt>
   0x0000555555555239 <+118>:   lea    -0x20(%rbp),%rdx
   0x000055555555523d <+122>:   lea    -0x60(%rbp),%rax
   0x0000555555555241 <+126>:   mov    %rdx,%rsi
   0x0000555555555244 <+129>:   mov    %rax,%rdi
   0x0000555555555247 <+132>:   call   0x555555555060 <strcmp@plt>
   0x000055555555524c <+137>:   test   %eax,%eax
   0x000055555555524e <+139>:   jne    0x55555555526d <main+170>
   0x0000555555555250 <+141>:   lea    -0x20(%rbp),%rax
   0x0000555555555254 <+145>:   mov    %rax,%rsi
   0x0000555555555257 <+148>:   lea    0xdc4(%rip),%rax        # 0x555555556022
   0x000055555555525e <+155>:   mov    %rax,%rdi
```

After the first breakpoint is hit, set a second one in the `deobfuscate` function

`break deobfuscate`

```
(gdb) break deobfuscate
Breakpoint 2 at 0x55555555517d
(gdb) c
Continuing.

Breakpoint 2, 0x000055555555517d in deobfuscate ()
```

The obfuscated flag is likely stored at `-0xa(%rbp)` as seen in:

```
0x00005555555551cb <+8>:     movabs $0x623f3f6f6c6b636d,%rax
0x00005555555551d5 <+18>:    mov    %rax,-0xa(%rbp)
```

The length of the flag is calculated dynamically using `strlen()`:

```
0x00005555555551f5 <+50>:    call   0x555555555040 <strlen@plt>
```

The length is stored in `%edx` before being passed to `deobfuscate()`.
The obfuscated flag and its length are passed to the `deobfuscate()` function:

```
0x0000555555555205 <+66>:    call   0x555555555179 <deobfuscate>
```

We can inspect the obfuscated flag and the length

```
(gdb) x/16xb $rbp-0xa
0x7fffffffdf86: 0x00    0x00    0xe0    0xe2    0xff    0xf7    0xff    0x7f
0x7fffffffdf8e: 0x00    0x00    0xe0    0xdf    0xff    0xff    0xff    0x7f
(gdb) print/x $edx
$1 = 0x9
```

Now we step in the function until we get to the XOR operation

```
(gdb) stepi
0x00005555555551ae in deobfuscate ()
=> 0x00005555555551ae <deobfuscate+53>: 32 45 fb                xor    -0x5(%rbp),%al
```

The obfuscated byte is located at `-0x5(%rbp)`.
The key being used for XOR is stored in `%al` (the lower 8 bits of the `rax` register).

Now we step in the function loop and read the value `x/s $rdi` until the full bytes are decrypted

At the end of the loop a `NOP` is performed, and reading the value we have the decrypted flag

```
(gdb) stepi
0x00005555555551b7 in deobfuscate ()
=> 0x00005555555551b7 <deobfuscate+62>: 8b 45 fc                mov    -0x4(%rbp),%eax
(gdb) stepi
0x00005555555551ba in deobfuscate ()
=> 0x00005555555551ba <deobfuscate+65>: 3b 45 e4                cmp    -0x1c(%rbp),%eax
(gdb) stepi
0x00005555555551bd in deobfuscate ()
=> 0x00005555555551bd <deobfuscate+68>: 7c d2                   jl     0x555555555191 <deobfuscate+24>
(gdb) stepi
0x00005555555551bf in deobfuscate ()
=> 0x00005555555551bf <deobfuscate+70>: 90                      nop
(gdb) x/s $rdi
0x7fffffffdfc0: "snakeCTF{79165ee8e}"
(gdb)
```

```
[redacted] ./ctf
Enter the correct flag: snakeCTF{79165ee8e}
Correct! The flag is: snakeCTF{79165ee8e}
```