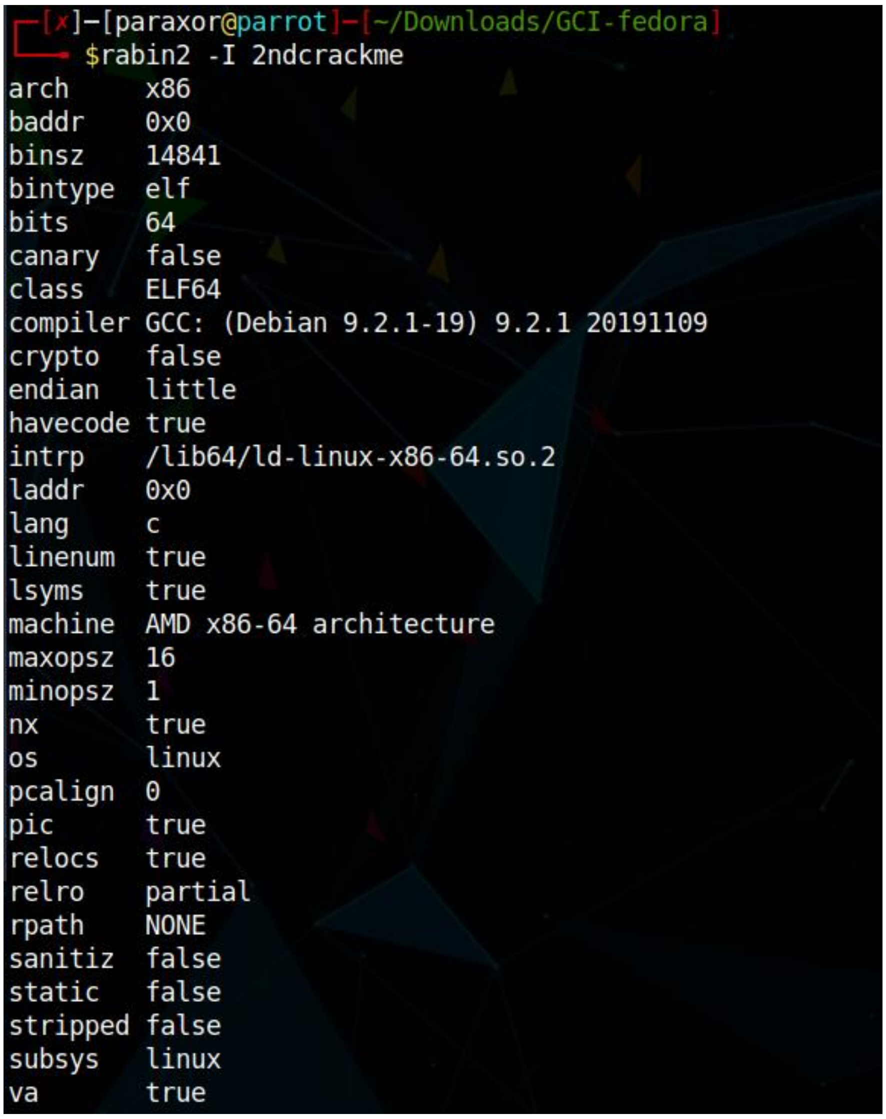
Reverse Engineering to extract password from binary files

2ndcrackme

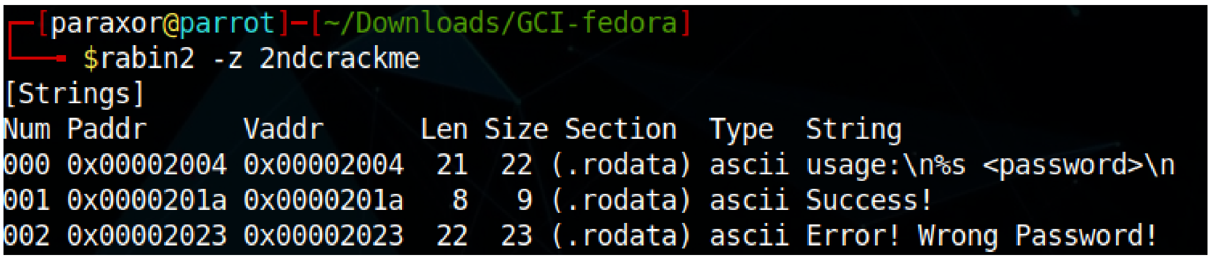
We've been given with a crack me file from which we must retrieve the password which is embedded in the memory. This file is a binary, so you can't retrieve it by normal means.

For basic knowledge, let us consider a main.c file. For a machine to understand the written code, the file is converted to an object file (.o extension) and further converted to a binary executable. To ensure the following, we do the following.



```
┌─[x]─[paraxor@parrot]─[~/Downloads/GCI-fedora]
└──╼ $rabin2 -I 2ndcrackme
arch     x86
baddr    0x0
binsz    14841
bintype  elf
bits     64
canary   false
class    ELF64
compiler GCC: (Debian 9.2.1-19) 9.2.1 20191109
crypto   false
endian   little
havecode true
intrp    /lib64/ld-linux-x86-64.so.2
laddr    0x0
lang     c
linenum  true
lsyms    true
machine  AMD x86-64 architecture
maxopsz  16
minopsz  1
nx       true
os       linux
pcalign  0
pic      true
relocs   true
relro    partial
rpath    NONE
sanitiz  false
static   false
stripped false
subsys   linux
va       true
```

I've played reverse engineering challenges before playing Google Code-In, so I have a set of processes in order to understand the binary.

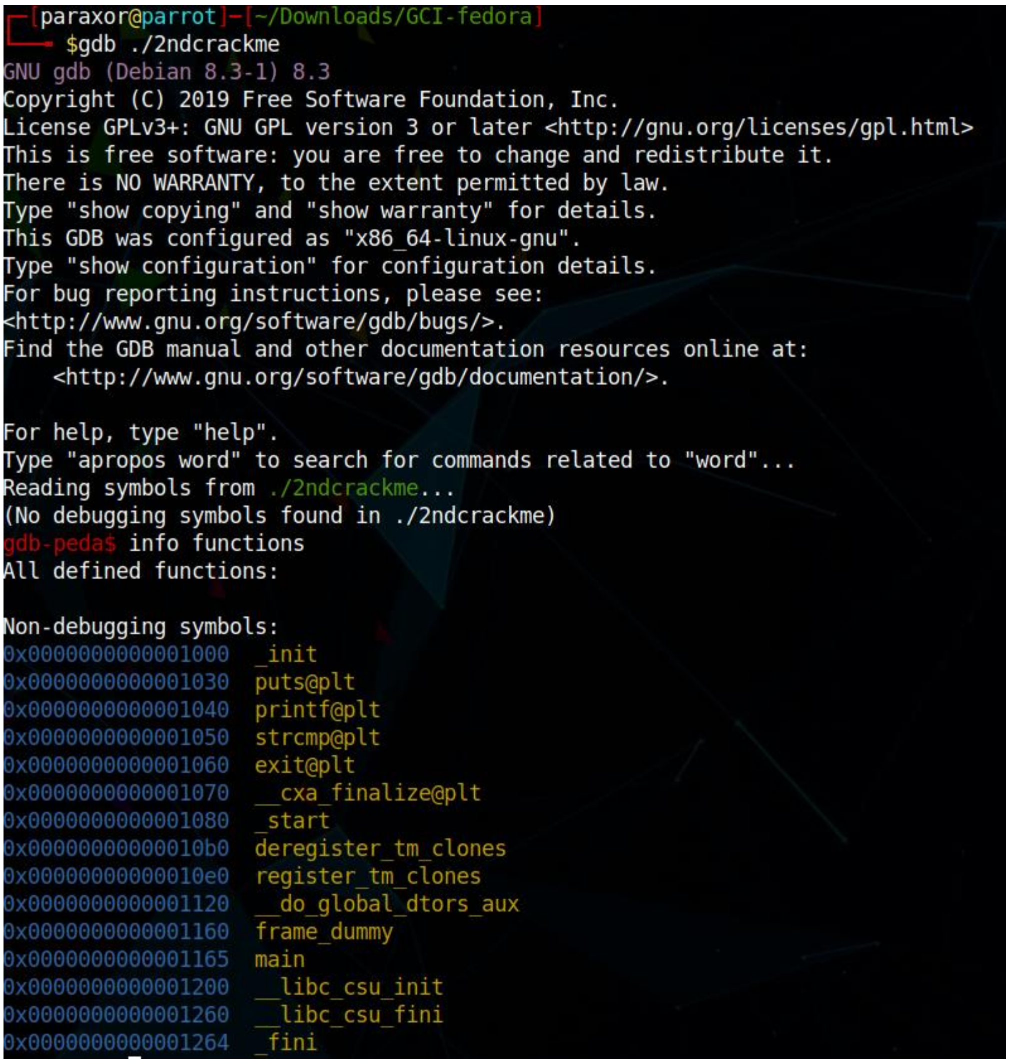


```
┌─[paraxor@parrot]─[~/Downloads/GCI-fedora]
└──╼ $rabin2 -z 2ndcrackme
[Strings]
Num Paddr      Vaddr      Len Size Section  Type  String
000 0x00002004 0x00002004  21  22 (.rodata) ascii usage:\n%s <password>\n
001 0x0000201a 0x0000201a   8   9 (.rodata) ascii Success!
002 0x00002023 0x00002023  22  23 (.rodata) ascii Error! Wrong Password!
```

Clever!

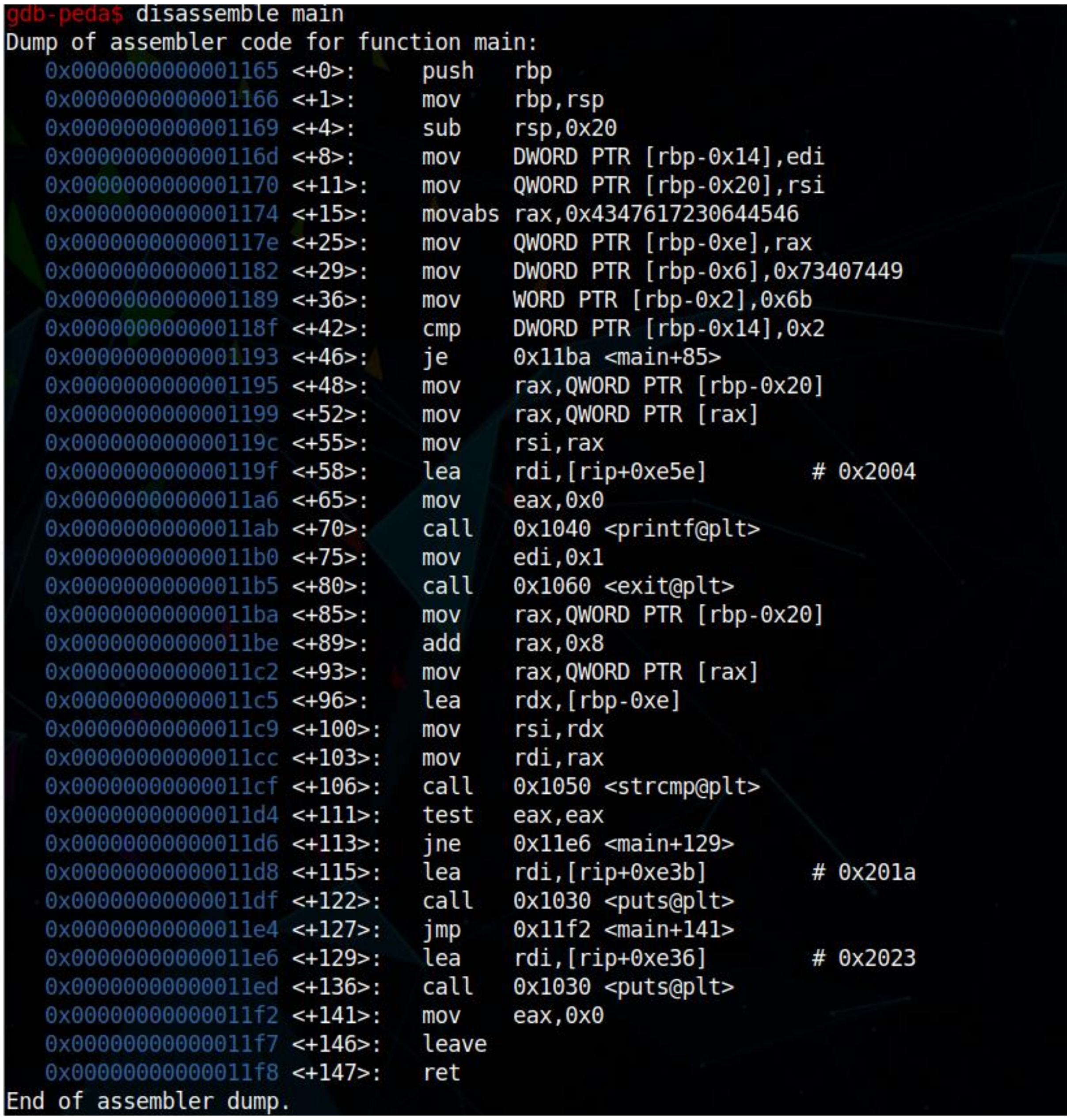
No more strings are hard coded here 😊 lets look deeper.

We use gdb-peda for the following. Gdb is a debugger and is inbuilt in Linux distributions. What I've used is an extension for gdb intended for binary exploitation purposes.



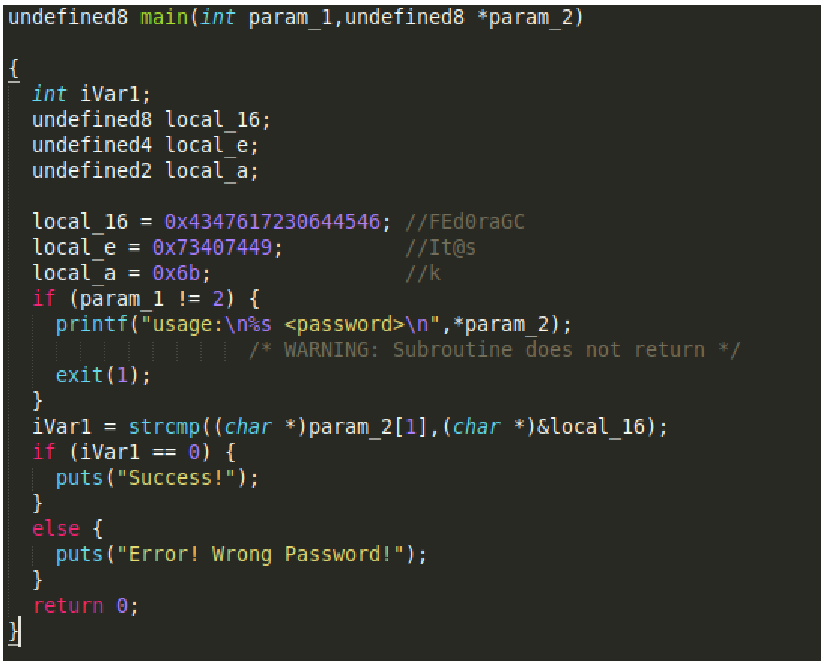
```
┌─[paraxor@parrot]─[~/Downloads/GCI-fedora]
└──╼ $gdb ./2ndcrackme
GNU gdb (Debian 8.3-1) 8.3
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./2ndcrackme...
(No debugging symbols found in ./2ndcrackme)
gdb-peda$ info functions
All defined functions:

Non-debugging symbols:
0x0000000000001000  _init
0x0000000000001030  puts@plt
0x0000000000001040  printf@plt
0x0000000000001050  strcmp@plt
0x0000000000001060  exit@plt
0x0000000000001070  __cxa_finalize@plt
0x0000000000001080  _start
0x00000000000010b0  deregister_tm_clones
0x00000000000010e0  register_tm_clones
0x0000000000001120  __do_global_dtors_aux
0x0000000000001160  frame_dummy
0x0000000000001165  main
0x0000000000001200  __libc_csu_init
0x0000000000001260  __libc_csu_fini
0x0000000000001264  _fini
```

We access the assembly code by the following.



```
gdb-peda$ disassemble main
Dump of assembler code for function main:
   0x0000000000001165 <+0>:	push   rbp
   0x0000000000001166 <+1>:	mov    rbp,rsp
   0x0000000000001169 <+4>:	sub    rsp,0x20
   0x000000000000116d <+8>:	mov    DWORD PTR [rbp-0x14],edi
   0x0000000000001170 <+11>:	mov    QWORD PTR [rbp-0x20],rsi
   0x0000000000001174 <+15>:	movabs rax,0x4347617230644546
   0x000000000000117e <+25>:	mov    QWORD PTR [rbp-0xe],rax
   0x0000000000001182 <+29>:	mov    DWORD PTR [rbp-0x6],0x73407449
   0x0000000000001189 <+36>:	mov    WORD PTR [rbp-0x2],0x6b
   0x000000000000118f <+42>:	cmp    DWORD PTR [rbp-0x14],0x2
   0x0000000000001193 <+46>:	je     0x11ba <main+85>
   0x0000000000001195 <+48>:	mov    rax,QWORD PTR [rbp-0x20]
   0x0000000000001199 <+52>:	mov    rax,QWORD PTR [rax]
   0x000000000000119c <+55>:	mov    rsi,rax
   0x000000000000119f <+58>:	lea    rdi,[rip+0xe5e]        # 0x2004
   0x00000000000011a6 <+65>:	mov    eax,0x0
   0x00000000000011ab <+70>:	call   0x1040 <printf@plt>
   0x00000000000011b0 <+75>:	mov    edi,0x1
   0x00000000000011b5 <+80>:	call   0x1060 <exit@plt>
   0x00000000000011ba <+85>:	mov    rax,QWORD PTR [rbp-0x20]
   0x00000000000011be <+89>:	add    rax,0x8
   0x00000000000011c2 <+93>:	mov    rax,QWORD PTR [rax]
   0x00000000000011c5 <+96>:	lea    rdx,[rbp-0xe]
   0x00000000000011c9 <+100>:	mov    rsi,rdx
   0x00000000000011cc <+103>:	mov    rdi,rax
   0x00000000000011cf <+106>:	call   0x1050 <strcmp@plt>
   0x00000000000011d4 <+111>:	test   eax,eax
   0x00000000000011d6 <+113>:	jne    0x11e6 <main+129>
   0x00000000000011d8 <+115>:	lea    rdi,[rip+0xe3b]        # 0x201a
   0x00000000000011df <+122>:	call   0x1030 <puts@plt>
   0x00000000000011e4 <+127>:	jmp    0x11f2 <main+141>
   0x00000000000011e6 <+129>:	lea    rdi,[rip+0xe36]        # 0x2023
   0x00000000000011ed <+136>:	call   0x1030 <puts@plt>
   0x00000000000011f2 <+141>:	mov    eax,0x0
   0x00000000000011f7 <+146>:	leave
   0x00000000000011f8 <+147>:	ret
End of assembler dump.
```

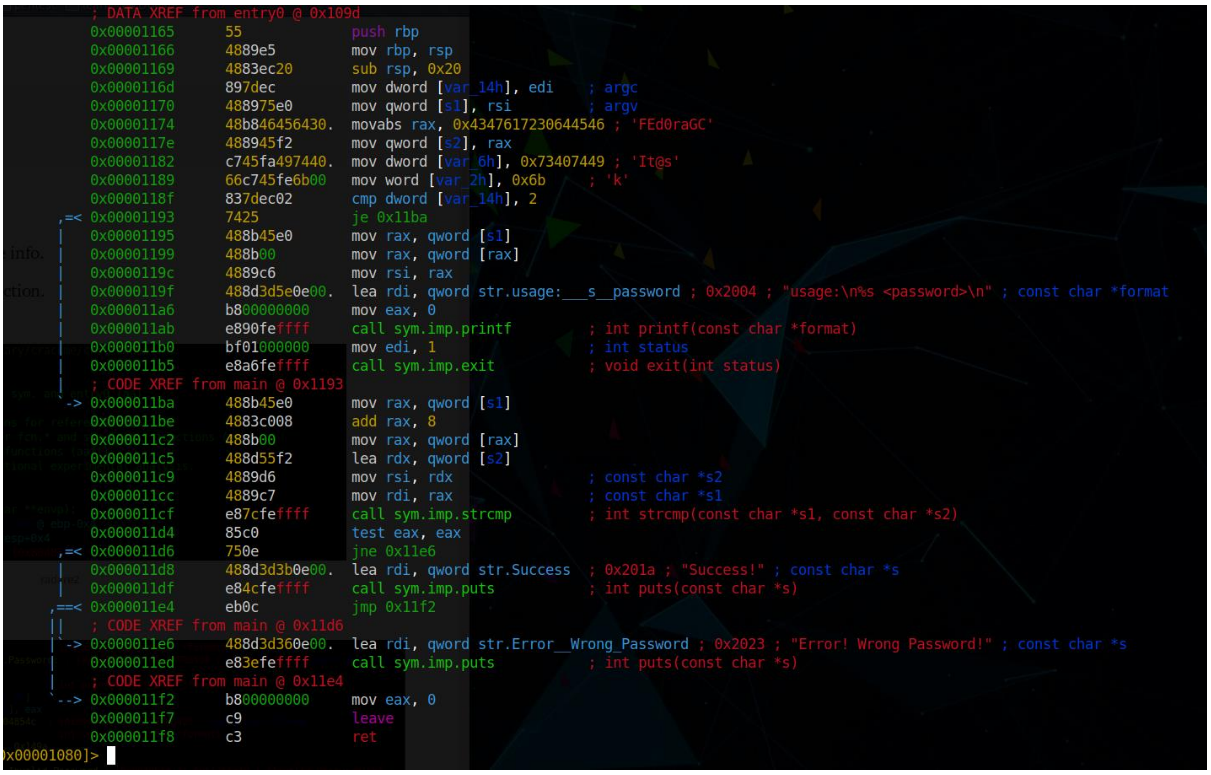
I didn’t want analyse the assembly code in detail, but sometimes you gotta do it.



```
undefined8 main(int param_1,undefined8 *param_2)

{
  int iVar1;
  undefined8 local_16;
  undefined4 local_e;
  undefined2 local_a;

  local_16 = 0x4347617230644546; //FEd0raGC
  local_e = 0x73407449;          //It@s
  local_a = 0x6b;                //k
  if (param_1 != 2) {
    printf("usage:\n%s <password>\n",*param_2);
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  iVar1 = strcmp((char *)param_2[1],(char *)&local_16);
  if (iVar1 == 0) {
    puts("Success!");
  }
  else {
    puts("Error! Wrong Password!");
  }
  return 0;
}
```

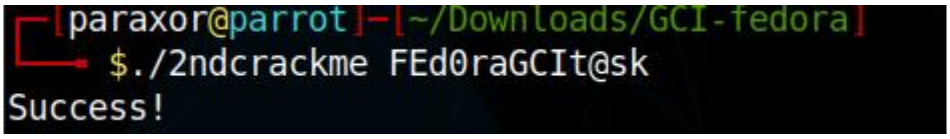
Encoded in hex huh? Not bad :p Lets look for more ways to solve this :v

I use r2 for most of the reverse engineering challenges. Lets look into it.



```
            ; DATA XREF from entry0 @ 0x109d
            0x00001165      55             push rbp
            0x00001166      4889e5         mov rbp, rsp
            0x00001169      4883ec20       sub rsp, 0x20
            0x0000116d      897dec         mov dword [var_14h], edi    ; argc
            0x00001170      488975e0       mov qword [s1], rsi         ; argv
            0x00001174      48b846456430.  movabs rax, 0x4347617230644546 ; 'FEd0raGC'
            0x0000117e      488945f2       mov qword [s2], rax
            0x00001182      c745fa497440.  mov dword [var_6h], 0x73407449 ; 'It@s'
            0x00001189      66c745fe6b00   mov word [var_2h], 0x6b     ; 'k'
            0x0000118f      837dec02       cmp dword [var_14h], 2
        ,=< 0x00001193      7425           je 0x11ba
        |   0x00001195      488b45e0       mov rax, qword [s1]
        |   0x00001199      488b00         mov rax, qword [rax]
        |   0x0000119c      4889c6         mov rsi, rax
        |   0x0000119f      488d3d5e0e00.  lea rdi, qword str.usage:___s__password ; 0x2004 ; "usage:\n%s <password>\n" ; const char *format
        |   0x000011a6      b800000000     mov eax, 0
        |   0x000011ab      e890feffff     call sym.imp.printf         ; int printf(const char *format)
        |   0x000011b0      bf01000000     mov edi, 1                  ; int status
        |   0x000011b5      e8a6feffff     call sym.imp.exit           ; void exit(int status)
        |   ; CODE XREF from main @ 0x1193
        `-> 0x000011ba      488b45e0       mov rax, qword [s1]
            0x000011be      4883c008       add rax, 8
            0x000011c2      488b00         mov rax, qword [rax]
            0x000011c5      488d55f2       lea rdx, qword [s2]
            0x000011c9      4889d6         mov rsi, rdx                ; const char *s2
            0x000011cc      4889c7         mov rdi, rax                ; const char *s1
            0x000011cf      e87cfeffff     call sym.imp.strcmp         ; int strcmp(const char *s1, const char *s2)
            0x000011d4      85c0           test eax, eax
        ,=< 0x000011d6      750e           jne 0x11e6
        |   0x000011d8      488d3d3b0e00.  lea rdi, qword str.Success  ; 0x201a ; "Success!" ; const char *s
        |   0x000011df      e84cfeffff     call sym.imp.puts           ; int puts(const char *s)
       ,==< 0x000011e4      eb0c           jmp 0x11f2
       ||   ; CODE XREF from main @ 0x11d6
       |`-> 0x000011e6      488d3d360e00.  lea rdi, qword str.Error__Wrong_Password ; 0x2023 ; "Error! Wrong Password!" ; const char *s
       |    0x000011ed      e83efeffff     call sym.imp.puts           ; int puts(const char *s)
       |    ; CODE XREF from main @ 0x11e4
       `--> 0x000011f2      b800000000     mov eax, 0
            0x000011f7      c9             leave
            0x000011f8      c3             ret
x00001080]>
```

Aaaaannddd, there you go. Your password. Fed0raGCIt@sk.



```
┌─[paraxor@parrot]─[~/Downloads/GCI-fedora]
└──╼ $./2ndcrackme FEd0raGCIt@sk
Success!
```