Reverse Engineering to extract password from binary files

1stcrackme

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 1stcrackme
arch     x86
baddr    0x0
binsz    14849
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 1stcrackme
[Strings]
Num Paddr      Vaddr      Len Size Section  Type  String
000 0x00002004 0x00002004  16  17 (.rodata) ascii Enter password:
001 0x00002018 0x00002018  17  18 (.rodata) ascii FEDORAGCIPASSEASY
002 0x0000202a 0x0000202a   9  10 (.rodata) ascii Success!\r
003 0x00002034 0x00002034  23  24 (.rodata) ascii Error! Wrong password!\r
004 0x0000204c 0x0000204c   6   7 (.rodata) ascii 0x1337
005 0x00002053 0x00002053   8   9 (.rodata) ascii 0x133337
```

This was a straightforward challenge and giving the above strings as hit-and-trail for the program confirms the password. But this seems unethical, so we look at the x86 assembly code.

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.

```
Parrot Terminal
File  Edit  View  Search  Terminal  Help
  -i, --info                 List object formats and architectures supported
  -H, --help                 Display this information
┌─[✗]─[paraxor@parrot]─[~/Downloads/GCI-fedora]
└──╼ $gdb ./1stcrackme
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 ./1stcrackme...
(No debugging symbols found in ./1stcrackme)
gdb-peda$ ls
1stcrackme  2ndcrackme  3rdcrackme
gdb-peda$ info functions
All defined functions:

Non-debugging symbols:
0x0000000000001000  _init
0x0000000000001030  puts@plt
0x0000000000001040  printf@plt
0x0000000000001050  strcmp@plt
0x0000000000001060  __isoc99_scanf@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
0x0000000000001290  __libc_csu_init
0x00000000000012f0  __libc_csu_fini
0x00000000000012f4  _fini
gdb-peda$ 
```

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>:	add    rsp,0xffffffffffffff80
   0x000000000000116d <+8>:	mov    DWORD PTR [rbp-0x74],edi
   0x0000000000001170 <+11>:	mov    QWORD PTR [rbp-0x80],rsi
   0x0000000000001174 <+15>:	lea    rdi,[rip+0xe89]        # 0x2004
   0x000000000000117b <+22>:	mov    eax,0x0
   0x0000000000001180 <+27>:	call   0x1040 <printf@plt>
   0x0000000000001185 <+32>:	lea    rax,[rbp-0x70]
   0x0000000000001189 <+36>:	mov    rsi,rax
   0x000000000000118c <+39>:	lea    rdi,[rip+0xe82]        # 0x2015
   0x0000000000001193 <+46>:	mov    eax,0x0
   0x0000000000001198 <+51>:	call   0x1060 <__isoc99_scanf@plt>
   0x000000000000119d <+56>:	lea    rax,[rbp-0x70]
   0x00000000000011a1 <+60>:	lea    rsi,[rip+0xe70]        # 0x2018
   0x00000000000011a8 <+67>:	mov    rdi,rax
   0x00000000000011ab <+70>:	call   0x1050 <strcmp@plt>
   0x00000000000011b0 <+75>:	test   eax,eax
   0x00000000000011b2 <+77>:	jne    0x11c2 <main+93>
   0x00000000000011b4 <+79>:	lea    rdi,[rip+0xe6f]        # 0x202a
   0x00000000000011bb <+86>:	call   0x1030 <puts@plt>
   0x00000000000011c0 <+91>:	jmp    0x11ce <main+105>
   0x00000000000011c2 <+93>:	lea    rdi,[rip+0xe6b]        # 0x2034
   0x00000000000011c9 <+100>:	call   0x1030 <puts@plt>
   0x00000000000011ce <+105>:	lea    rdi,[rip+0xe2f]        # 0x2004
   0x00000000000011d5 <+112>:	mov    eax,0x0
   0x00000000000011da <+117>:	call   0x1040 <printf@plt>
   0x00000000000011df <+122>:	lea    rax,[rbp-0x70]
   0x00000000000011e3 <+126>:	mov    rsi,rax
   0x00000000000011e6 <+129>:	lea    rdi,[rip+0xe28]        # 0x2015
   0x00000000000011ed <+136>:	mov    eax,0x0
   0x00000000000011f2 <+141>:	call   0x1060 <__isoc99_scanf@plt>
   0x00000000000011f7 <+146>:	lea    rax,[rbp-0x70]
   0x00000000000011fb <+150>:	lea    rsi,[rip+0xe4a]        # 0x204c
   0x0000000000001202 <+157>:	mov    rdi,rax
   0x0000000000001205 <+160>:	call   0x1050 <strcmp@plt>
```

This assembly code isn't complete, but one can write a brief decompiled code from the following. What we need to stress on is the <strcmp@plt> part which compares the following inputs.

```c
undefined8 main(void)

{
  int iVar1;
  char local_78 [112];

  printf("Enter password: ");
  __isoc99_scanf(&DAT_00102015,local_78);
  iVar1 = strcmp(local_78,"FEDORAGCIPASSEASY");
  if (iVar1 == 0) {
    puts("Success!\r");
  }
  else {
    puts("Error! Wrong password!\r");
  }
  printf("Enter password: ");
  __isoc99_scanf(&DAT_00102015,local_78);
  iVar1 = strcmp(local_78,"0x1337");
  if (iVar1 == 0) {
    puts("Success!\r");
  }
  else {
    puts("Error! Wrong password!\r");
  }
  printf("Enter password: ");
  __isoc99_scanf(&DAT_00102015,local_78);
  iVar1 = strcmp(local_78,"0x133337");
  if (iVar1 == 0) {
    puts("Success!\r");
  }
  else {
    puts("Error! Wrong password!\r");
  }
  return 0;
}
```

As you see, there are three passwords which satisfy the given crackme file. But these instructions should be given in order for calling puts("Success!\r");

And this ensures the following.

```
┌─[paraxor@parrot]─[~/Downloads/GCI-fedora]
└──╼ $./1stcrackme
Enter password: FEDORAGCIPASSEASY
Success!
Enter password: 0x1337
Success!
Enter password: 0x133337
Success!
```

1stcrackme done! :)