TryHackMe: Reversing ELF

The following writeup is for [Reversing ELF]() on TryHackMe, it involves reverse engineering a series of ELF files.

Crackme1

The first challenge is simple, all we need to do is execute the binary:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ chmod +x crackme1

┌──(kali㉿kali)-[~/Downloads]
└─$ ./crackme1
flag{not_that_kind_of_elf}
```

Crackme2

This crackme involves finding a password in the binary, and using that to obtain the flag:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./crackme2
Usage: ./crackme2 password
```

I am going to use radare2 which is preinstalled in Kali Linux. To launch radare2 and starting analysing the binary, we can enter the following command:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ r2 ./crackme2
```

To analyse the binary, use the aa command:

```
[0x080483a0]> aa
INFO: Analyze all flags starting with sym. and entry0 (aa)
INFO: Analyze imports (af@@@i)
INFO: Analyze entrypoint (af@ entry0)
INFO: Analyze symbols (af@@@s)
INFO: Recovering variables (afva@@@F)
INFO: Analyze all functions arguments/locals (afva@@@F)
```

If we list the function within this binary, we can see a main function:

```
[0x080483a0]> afl
0x08048340    1      6 sym.imp.strcmp
0x08048350    1      6 sym.imp.printf
0x08048360    1      6 sym.imp.puts
0x08048370    1      6 sym.imp.__libc_start_main
0x08048380    1      6 sym.imp.memset
0x080483a0    1     33 entry0
0x080483e0    4     43 sym.deregister_tm_clones
0x08048410    4     53 sym.register_tm_clones
0x08048450    3     30 entry.fini0
0x08048470    4     40 entry.init0
0x08048526    4    149 sym.giveFlag
0x08048620    1      2 sym.__libc_csu_fini
0x080483d0    1      4 sym.__x86.get_pc_thunk.bx
0x08048624    1     20 sym._fini
0x080485c0    4     93 sym.__libc_csu_init
0x0804849b    6    139 main
0x08048304    3     35 sym._init
0x08048390    1      6 fcn.08048390
```

Let’s go explore this main function to see if we can determine the logic for this binary:

```
[0x080483a0]> pdf @ main
            ; DATA XREF from entry0 @ 0x80483b7(w)
┌ 139: int main (char **argv);
│           ; var int32_t var_4h @ ebp-0x4
│           ; arg char **argv @ esp+0x24
│           0x0804849b      8d4c2404       lea ecx, [argv]
│           0x0804849f      83e4f0         and esp, 0xfffffff0
│           0x080484a2      ff71fc         push dword [ecx - 4]
│           0x080484a5      55             push ebp
│           0x080484a6      89e5           mov ebp, esp
│           0x080484a8      51             push ecx
│           0x080484a9      83ec04         sub esp, 4
│           0x080484ac      89c8           mov eax, ecx
│           0x080484ae      833802         cmp dword [eax], 2
│       ┌─< 0x080484b1      741d           je 0x80484d0
│       │   0x080484b3      8b4004         mov eax, dword [eax + 4]
│       │   0x080484b6      8b00           mov eax, dword [eax]
│       │   0x080484b8      83ec08         sub esp, 8
│       │   0x080484bb      50             push eax
│       │   0x080484bc      6860860408     push str.Usage:__s_password_n ; 0x8048660 ; "Usage: %s password\n"
│       │   0x080484c1      e88afeffff     call sym.imp.printf         ; int printf(const char *format)
│       │   0x080484c6      83c410         add esp, 0x10
│       │   0x080484c9      b801000000     mov eax, 1
│      ┌──< 0x080484ce      eb4e           jmp 0x804851e
│      │└─> 0x080484d0      8b4004         mov eax, dword [eax + 4]
│      │    0x080484d3      83c004         add eax, 4
│      │    0x080484d6      8b00           mov eax, dword [eax]
│      │    0x080484d8      83ec08         sub esp, 8
│      │    0x080484db      6874860408     push str.super_secret_password ; 0x8048674 ; "super_secret_password"
│      │    0x080484e0      50             push eax
│      │    0x080484e1      e85afeffff     call sym.imp.strcmp         ; int strcmp(const char *s1, const char *s2)
│      │    0x080484e6      83c410         add esp, 0x10
│      │    0x080484e9      85c0           test eax, eax
│      │┌─< 0x080484eb      7417           je 0x8048504
│      ││   0x080484ed      83ec0c         sub esp, 0xc
│      ││   0x080484f0      688a860408     push str.Access_denied.     ; 0x804868a ; "Access denied."
│      ││   0x080484f5      e866feffff     call sym.imp.puts           ; int puts(const char *s)
│      ││   0x080484fa      83c410         add esp, 0x10
│      ││   0x080484fd      b801000000     mov eax, 1
│     ┌───< 0x08048502      eb1a           jmp 0x804851e
│     ││└─> 0x08048504      83ec0c         sub esp, 0xc
│     ││    0x08048507      6899860408     push str.Access_granted.    ; 0x8048699 ; "Access granted."
│     ││    0x0804850c      e84ffeffff     call sym.imp.puts           ; int puts(const char *s)
│     ││    0x08048511      83c410         add esp, 0x10
│     ││    0x08048514      e80d000000     call sym.giveFlag
│     ││    0x08048519      b800000000     mov eax, 0
│     ││    ; CODE XREFS from main @ 0x80484ce(x), 0x8048502(x)
│     └└──> 0x0804851e      8b4dfc         mov ecx, dword [var_4h]
│           0x08048521      c9             leave
│           0x08048522      8d61fc         lea esp, [ecx - 4]
└           0x08048525      c3             ret
```

At a quick glance, it is pretty obvious that the password is “super_secret_password”. Let’s confirm this by running the binary and supplying this password:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./crackme2 super_secret_password
Access granted.
flag{if_i_submit_this_flag_then_i_will_get_points}
```

We know that the password is super_secret_password due to the following block of instructions:

```
push str.super_secret_password ; 0×8048674 ; "super_secret_password"
push eax
call sym.imp.strcmp           ; int strcmp(const char *s1, const char *s2)
```

This is comparing the second argument to the hardcoded string password using strcmp. In all honestly, you could have just listed the strings within the binary to find the password:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ strings crackme2
/lib/ld-linux.so.2
libc.so.6
_IO_stdin_used
puts
printf
memset
strcmp
__libc_start_main
/usr/local/lib:$ORIGIN
__gmon_start__
GLIBC_2.0
PTRh
j3jA
[^_]
UWVS
t$,U
[^_]
Usage: %s password
super_secret_password
```

Crackme3

If we analyse the main function of this binary like we did before, we can find a very interesting string:

```
mov dword [var_4h], str.ZjByX3kwdXJfNWVjMG5kX2xlNTVvbl91bmJhc2U2NF80bGxfN2gzXzdoMW5nNQ ; [0×8048e8b:4]=0×79426a5a ; "ZjByX3kwdXJfNWVjMG5kX2xlNTVvbl91bmJhc2U2NF80bGxfN2gzXzdoMW5nNQ=="
call sym.imp.strcmp          ; int strcmp(const char *s1, const char *s2)
test eax, eax
jne 0×80485b8
mov dword [esp], str.Correct_password_ ; [0×8048ed0:4]=0×72726f43 ; "Correct password!"
```

This appears to be base64 encoded text. Once again, we can see the strcmp function, which likely indicates that this encoded string is the password. If however, we just supply this encoded string as the password, we wont find the flag:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./crackme3 ZjByX3kwdXJfNWVjMG5kX2xlNTVvbl91bmJhc2U2NF80bGxfN2gzXzdoMW5nNQ==
Come on, even my aunt Mildred got this one!
```

First, we need to decode the string and supply the decoded text, we can do so in the command line:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ echo "ZjByX3kwdXJfNWVjMG5kX2xlNTVvbl91bmJhc2U2NF80bGxfN2gzXzdoMW5nNQ==" | base64 -d
f0r_y0ur_5ec0nd_le55on_unbase64_4ll_7h3_7h1ng5
```

```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./crackme3 f0r_y0ur_5ec0nd_le55on_unbase64_4ll_7h3_7h1ng5
Correct password!
```

Crackme4

This crackme was far more difficult compared to the previous ones. If we look at the main function, we can see it calling a function called sym.compare_pwd:

```
[0x00400540]> pdf @ main
            ; DATA XREF from entry0 @ 0x40055d(r)
┌ 74: int main (int argc, char **argv);
│           ; arg int argc @ rdi
│           ; arg char **argv @ rsi
│           ; var int64_t var_4h @ rbp-0x4
│           ; var int64_t var_10h @ rbp-0x10
│           0x00400716      55             push rbp
│           0x00400717      4889e5         mov rbp, rsp
│           0x0040071a      4883ec10       sub rsp, 0x10
│           0x0040071e      897dfc         mov dword [var_4h], edi      ; argc
│           0x00400721      488975f0       mov qword [var_10h], rsi     ; argv
│           0x00400725      837dfc02       cmp dword [var_4h], 2
│       ┌─< 0x00400729      741b           je 0x400746
│       │   0x0040072b      488b45f0       mov rax, qword [var_10h]
│       │   0x0040072f      488b00         mov rax, qword [rax]
│       │   0x00400732      4889c6         mov rsi, rax
│       │   0x00400735      bf10084000     mov edi, str.Usage_:__s_password_nTh
│       │   0x0040073a      b800000000     mov eax, 0
│       │   0x0040073f      e8bcfdffff     call sym.imp.printf          ; int pr
│      ┌──< 0x00400744      eb13           jmp 0x400759
│      │└─> 0x00400746      488b45f0       mov rax, qword [var_10h]
│      │    0x0040074a      4883c008       add rax, 8
│      │    0x0040074e      488b00         mov rax, qword [rax]
│      │    0x00400751      4889c7         mov rdi, rax
│      │    0x00400754      e821ffffff     call sym.compare_pwd
```

If we check out this function, we can see what appears to be 3 separate variables:

```
movabs rax, 0x7b175614497b5d49 ; 'I]{I\x14V\x17{'
mov qword [var_20h], rax
movabs rax, 0x547b175651474157 ; 'WAGQV\x17{T'
mov qword [var_18h], rax
mov word [var_10h], 0x4053  ; 'S@'
```

We also see a call to sym.get_pwd:

```
call sym.get_pwd
```

In this function, the strings we found earlier (and believe to form the password) are XORed with 0x24:

```
xor eax, 0×24
```

If we decode these strings, we are given the following password: my_m0r3_secur3_pwd

```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./crackme4 my_m0r3_secur3_pwd
password OK
```

Decoding process was like as follows:

Byte (Hex)	XOR with 0x24	Result (Hex)	ASCII
0x49	0x49 ^ 0x24	0x6D	m
0x5D	0x5D ^ 0x24	0x79	y
0x7B	0x7B ^ 0x24	0x5F	_
0x49	0x49 ^ 0x24	0x6D	m
0x14	0x14 ^ 0x24	0x30	0
0x56	0x56 ^ 0x24	0x72	r
0x17	0x17 ^ 0x24	0x33	3
0x7B	0x7B ^ 0x24	0x5F	_

Result: my_m0r3_

var_18h: 0x547b175651474157

Byte (Hex)	XOR with 0x24	Result (Hex)	ASCII
0x57	0x57 ^ 0x24	0x73	s
0x41	0x41 ^ 0x24	0x65	e
0x47	0x47 ^ 0x24	0x63	c
0x51	0x51 ^ 0x24	0x75	u
0x56	0x56 ^ 0x24	0x72	r
0x17	0x17 ^ 0x24	0x33	3
0x7B	0x7B ^ 0x24	0x5F	_

Byte (Hex)	XOR with 0x24	Result (Hex)	ASCII
0x54	0x54 ^ 0x24	0x70	p

Result: secur3_p

var_10h: 0x4053

Byte (Hex)	XOR with 0x24	Result (Hex)	ASCII
0x53	0x53 ^ 0x24	0x77	w
0x40	0x40 ^ 0x24	0x64	d

Result: wd

Crackme5

This one was surprisingly easy. All we need to do is analyse the main function and concatenate each variable from var_30h to var_15h:

```
mov byte [var_30h], 0x4f    ; 'O' ; 79
mov byte [var_2fh], 0x66    ; 'f' ; 102
mov byte [var_2eh], 0x64    ; 'd' ; 100
mov byte [var_2dh], 0x6c    ; 'l' ; 108
mov byte [var_2ch], 0x44    ; 'D' ; 68
mov byte [var_2bh], 0x53    ; 'S' ; 83
mov byte [var_2ah], 0x41    ; 'A' ; 65
mov byte [var_29h], 0x7c    ; '|' ; 124
mov byte [var_28h], 0x33    ; '3' ; 51
mov byte [var_27h], 0x74    ; 't' ; 116
mov byte [var_26h], 0x58    ; 'X' ; 88
mov byte [var_25h], 0x62    ; 'b' ; 98
mov byte [var_24h], 0x33    ; '3' ; 51
mov byte [var_23h], 0x32    ; '2' ; 50
mov byte [var_22h], 0x7e    ; '~' ; 126
mov byte [var_21h], 0x58    ; 'X' ; 88
mov byte [var_20h], 0x33    ; '3' ; 51
mov byte [var_1fh], 0x74    ; 't' ; 116
mov byte [var_1eh], 0x58    ; 'X' ; 88
mov byte [var_1dh], 0x40    ; elf_phdr
mov byte [var_1ch], 0x73    ; 's' ; 115
mov byte [var_1bh], 0x58    ; 'X' ; 88
mov byte [var_1ah], 0x60    ; '`' ; 96
mov byte [var_19h], 0x34    ; '4' ; 52
mov byte [var_18h], 0x74    ; 't' ; 116
mov byte [var_17h], 0x58    ; 'X' ; 88
mov byte [var_16h], 0x74    ; 't' ; 116
mov byte [var_15h], 0x7a    ; 'z' ; 122
```

Therefore, the answer/input is OfdlDSA|3tXb32~X3tX@sX` 4tXtz

Crackme6

This crackme was similar to crackme4. If we analyse the main function, we can see that it calls a function called sym.compare_pwd:

```
call sym.compare_pwd
```

In sym.compare_pwd, sym.my_secure_test is called so lets check this function out. In sym.my_secure_test, it checks whether a given input matches a hardcoded sequence of characters. In all honesty, I had no idea how to decode this, but thankfully ChatGPT did:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./crackme6 1337_pwd

password OK
```

Therefore, the answer is 1337_pwd

Crackme7

With this binary, we are presented a menu of sorts which is stored in var_ch. It has various cases depending on the user's input. We are most concerned about this line:

```
cmp eax, 0x7a69                ; 'iz'
```

If we enter the value 31337 (decimal), it calls the giveFlag function and gives us the flag:

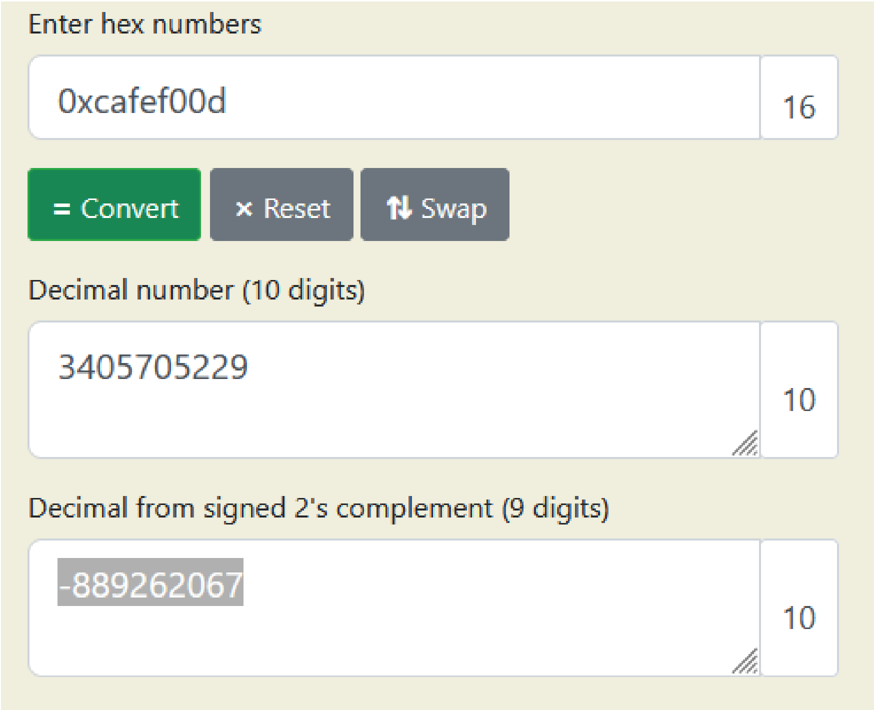
```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./crackme7
Menu:

[1] Say hello
[2] Add numbers
[3] Quit

[>] 31337
Wow such h4x0r!
flag{much_reversing_very_ida_wow}
```

Crackme8

If we analyse the main function (or debug the program), there is a cmp with 0xcafef00d. If we convert this to decimal from signed 2's component, we are given the following:



If you enter this value, you are given the flag:

```
┌──(kali㉿kali)-[~/Downloads]
└─$ ./crackme8 -889262067
Access granted.
flag{at_least_this_cafe_wont_leak_your_credit_card_numbers}
```

This room was my first experience reverse engineering ELF binaries, and really reverse engineering in general. If you are new to reverse engineering like myself, I highly recommend doing this CTF and using tools like radare2 and ChatGPT.