



# (Write-up) THM: Reversing ELF with Radare2

Note: Sorry if there are any misspelled words in the write-up (I am not a native English speaker).

## TryHackMe | Reversing ELF

TryHackMe is an online platform for learning and teaching cyber security, all through your browser.



<https://tryhackme.com/room/reverselffiles>



## Crackme1

To get the first flag we just need to run the binary. But first we need to give it permissions to run with `chmod +x crackme1`.

```
~/tryhackme/reversing ➔ ./crackme1
flag{[REDACTED]}
~/tryhackme/reversing ➔
```

## Crackme2

The second crackme asks for a password as an argument. If the password is wrong it outputs "Access denied".

```
~/tryhackme/reversing ➤ ./crackme2
Usage: ./crackme2 password
✖ ~/tryhackme/reversing ➤ ./crackme2 aaaaa
Access denied.
```

Running the strings command reveals us the super secret password.

```
x ~ /tryhackme/reversing ➤ 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
SL
Access denied.
Access granted.
;*2$"(

GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609
crtstuff.c
__JCR_LIST__
deregister_tm_clones
__do_global_dtors_aux
completed.7209
__do_global_dtors_aux_fini_array_entry
frame_dummy
__frame_dummy_init_array_entry
conditional1.c
giveFlag
__FRAME_END__
__JCR_END__
```

Now we can run the binary with the super secret password as argument to get the flag.

```
~/tryhackme/reversing ➤ ./crackme2
Access granted.
flag{[REDACTED]}

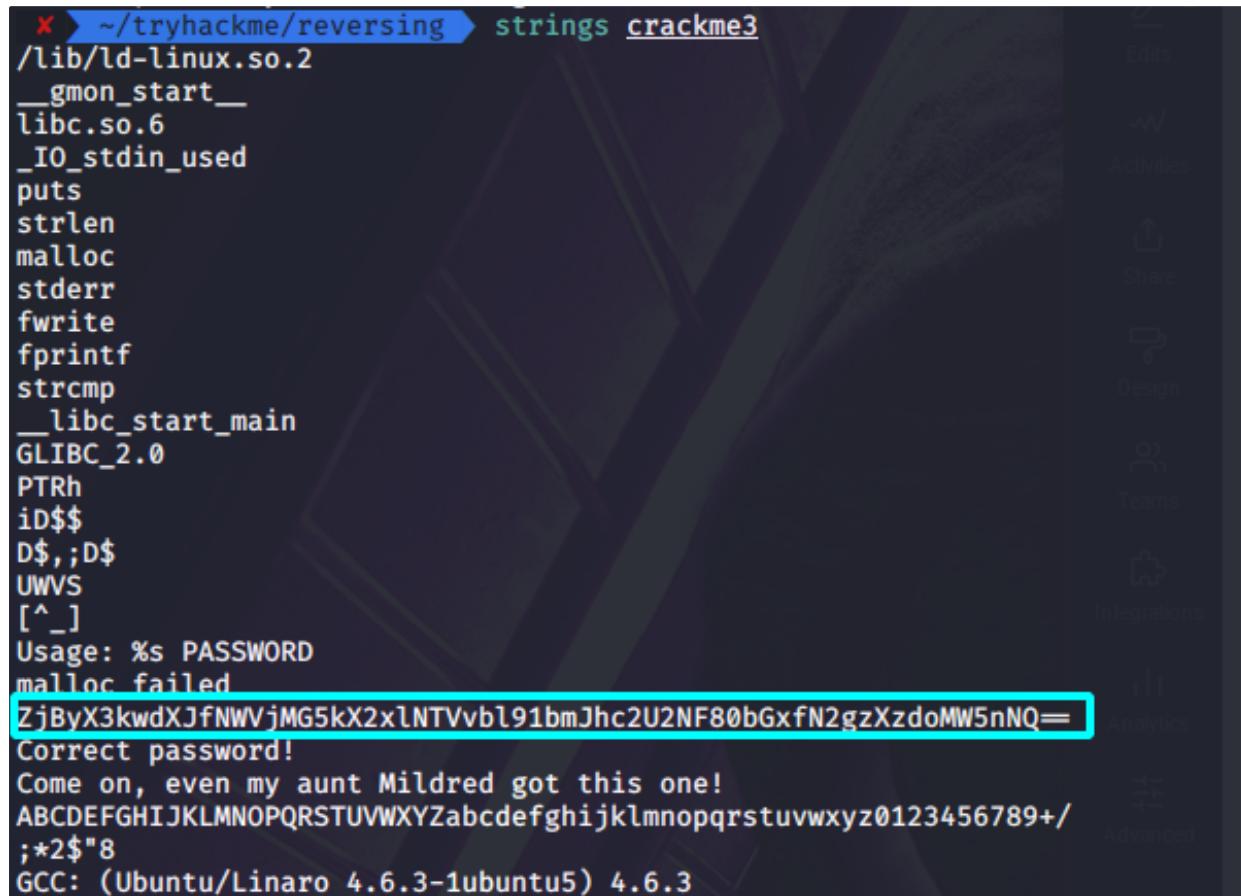
~/tryhackme/reversing ➤
```

## Crackme3

This crackme, as the previous one, asks for the right password as an argument.

```
~/tryhackme/reversing ➤ ./crackme3
Usage: ./crackme3 PASSWORD
✗ ➤ ~/tryhackme/reversing ➤ ./crackme3 password
Come on, even my aunt Mildred got this one!
✗ ➤ ~/tryhackme/reversing ➤
```

Running the strings command on the binary reveals a base64 encoded string.



```
x ~/tryhackme/reversing ➤ strings crackme3
/lib/ld-linux.so.2
__gmon_start__
libc.so.6
_IO_stdin_used
puts
strlen
malloc
stderr
fwrite
fprintf
strcmp
__libc_start_main
GLIBC_2.0
PTRh
iD$$
D$,;D$
UWVS
[^_]
Usage: %s PASSWORD
malloc failed
ZjByX3kwdXJfNWVjMG5kX2x1NTVvb191bmJhc2U2NF80bGxfN2gzXzdoMW5nNQ==
Correct password!
Come on, even my aunt Mildred got this one!
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
;*2$"8
GCC: (Ubuntu/Linaro 4.6.3-1ubuntu5) 4.6.3
```

Decoding the string gives us the flag.

```
~/tryhackme/reversing ➤ echo "ZjByX3kwdXJfNWVjMG5kX2x1NTVvb191bmJhc2U2NF80bGxfN2gzXzdoMW5nNQ==" | base64 --decode
f
~/tryhackme/reversing ➤
```

## Crackme4

This crackme is similar to the previous ones, where you need to provide a password as an argument, but this time, when we run it, it outputs a message saying that the string is hidden and that it uses strcmp.

```
~/tryhackme/reversing ➤ ./crackme4
Usage: ./crackme4 password
This time the string is hidden and we used strcmp
~/tryhackme/reversing ➤ ./crackme4 password
password "password" not OK
~/tryhackme/reversing ➤
```

If we see the man page of strcmp, with the command `man strcmp`, we can see in its description that it compares two strings and returns 0 if they are equal.

**DESCRIPTION**  
The `strcmp()` function compares the two strings `s1` and `s2`. The locale is not taken into account (for a locale-aware comparison, see `strcoll(3)`). The comparison is done using unsigned characters.  
Kali Linux  
amd64  
`strcmp()` returns an integer indicating the result of the comparison, as follows:  
• 0, if the `s1` and `s2` are equal;  
• a negative value if `s1` is less than `s2`;  
• a positive value if `s1` is greater than `s2`;  
The `strncmp()` function is similar, except it compares only the first (at most) `n` bytes of `s1` and `s2`.

To solve this and the rest of the crackmes I will use `radare2`.

We can start the binary in debug mode with radare using the command `r2 -d crackme4`. Then we analyze the binary with the command `aaa` and check for its functions with `afl`.

```
~/tryhackme/reversing ➤ r2 -d crackme4
Process with PID 3304 started ...
= attach 3304 3304
bin.baddr 0x00400000
Using 0x400000
asm.bits 64
Warning: r_bin_file_hash: file exceeds bin.hashlimit
[0x7f2b1a81c090]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[TOFIX: aaft can't run in debugger mode.ions (aaft)
[x] Type matching analysis for all functions (aaft)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x7f2b1a81c090]> afl
0x00400540    1 41          entry0
0x00400510    1 6           sym.imp.__libc_start_main
0x00400570    4 41          sym.deregister_tm_clones
0x004005a0    4 57          sym.register_tm_clones
0x004005e0    3 28          entry.fini0
0x00400600    4 45      → 42 entry.init0
0x004007d0    1 2           sym.__libc_csu_fini
0x0040062d    4 77          sym.get_pwd
0x004007d4    1 9           sym._fini
0x0040067a    6 156          sym.compare_pwd
0x00400760    4 101         sym.__libc_csu_init
0x00400716    4 74          main
0x004004b0    3 26          sym._init
0x00400530    1 6           loc.imp.__gmon_start
0x004004e0    1 6           sym.imp.puts
0x004004f0    1 6           sym.imp.__stack_chk_fail
0x00400500    1 6           sym.imp.printf
0x00400520    1 6           sym.imp.strncmp
[0x7f2b1a81c090]> █
```

We see that the binary runs a lot of function, but the most important here and in most binaries is the function `main`. We can analyze the main function running `pdf @main`.

```
[0x7f2b1a81c090]> pdf @main
    ; DATA XREF from entry0 @ 0x40055d
74: int main (int argc, char **argv, char **envp);
    ; var int64_t var_10h @ rbp-0x10
    ; var int64_t var_4h @ rbp-0x4
    ; arg int argc @ rdi
    ; arg char **argv @ rsi
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_This_time_the_string_is_hidden_and_we
used_strcmp ; 0x400810 ; "Usage : %s password\nThis time the string is hidden and we used strcmp\n"
0x0040073a      b800000000 mov eax, 0
0x0040073f      e8bcfdffff call sym.imp.printf      ; int printf(const char *format)
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      e821fffffe call sym.compare_pwd
; CODE XREF from main @ 0x400744
0x00400759      b800000000 mov eax, 0
0x0040075e      c9      leave
0x0040075f      c3      ret
[0x7f2b1a81c090]>
```

Analyzing the main function shows us a lot of assembly code that we can ignore for now, what is important for us is that the main function calls another function, `sym.compare_pwd`, before it returns. We can analyze this function with the command `pdf @sym.compare_pwd`.

```

[0x7f2b1a81c090]> pdf @sym.compare_pwd
      ; CALL XREF from main @ 0x400754
156: sym.compare_pwd (int64_t arg1);
      ; var int64_t var_28h @ rbp-0x28
      ; var int64_t var_20h @ rbp-0x20
      ; var int64_t var_18h @ rbp-0x18
      ; var int64_t var_10h @ rbp-0x10
      ; var int64_t var_eh @ rbp-0xe
      ; var int64_t var_8h @ rbp-0x8
      ; arg int64_t arg1 @ rdi
0x0040067a      55          push rbp
0x0040067b      4889e5      mov rbp, rsp
0x0040067e      4883ec30    sub rsp, 0x30
0x00400682      488970d8    mov qword [var_28h], rdi    ; arg1
0x00400686      64488b042528. mov rax, qword fs:[0x28]
0x0040068f      488945f8    mov qword [var_8h], rax
0x00400693      31c0        xor eax, eax
0x00400695      48b8495d7b49. movabs rax, 0x7b175614497b5d49
0x0040069f      488945e0    mov qword [var_20h], rax
0x004006a3      48b857414751. movabs rax, 0x547b175651474157
0x004006ad      488945e8    mov qword [var_18h], rax
0x004006b1      66c745f05340  mov word [var_10h], 0x4053 ; 'S@'
0x004006b7      c645f200    mov byte [var_eh], 0
0x004006bb      488d45e0    lea rax, qword [var_20h]
0x004006bf      4889c7      mov rdi, rax
0x004006c2      e866ffffff  call sym.get_pwd
0x004006c7      488b55d8    mov rdx, qword [var_28h]
0x004006cb      488d45e0    lea rax, qword [var_20h]
0x004006cf      4889d6      mov rsi, rdx
0x004006d2      4889c7      mov rdi, rax
0x004006d5      e846ffffff  call sym.imp.strptime ; int strcmp(const char *s1, const char *s2)

      < 0x004006da      85c0        test eax, eax
      < 0x004006dc      750c        jne 0x4006ea
      < 0x004006de      bfe8074000  mov edi, str.password_OK ; 0x4007e8 ; "password OK"
      < 0x004006e3      e8f8fdffff  call sym.imp.puts ; int puts(const char *s)
      < 0x004006e8      eb16        jmp 0x400700
      < 0x004006ea      488b45d8    mov rax, qword [var_28h]
      < 0x004006ee      4889c6      mov rsi, rax
      < 0x004006f1      bff4074000  mov edi, str.password__s__not_OK ; 0x4007f4 ; "password \"%s\" not OK"

      < 0x004006f6      b800000000  mov eax, 0
      < 0x004006fb      e800ffff    call sym.imp.printf ; int printf(const char *format)
      ; CODE XREF from sym.compare_pwd @ 0x4006e8
      > 0x00400700      488b45f8    mov rax, qword [var_8h]
      > 0x00400704      644833042528. xor rax, qword fs:[0x28]
      > 0x0040070d      7405        je 0x400714
      > 0x0040070f      e8dcfdffff  call sym.imp.__stack_chk_fail ; void __stack_chk_fail(void)
      > 0x00400714      c9          leave
      > 0x00400715      c3          ret

[0x7f2b1a81c090]>

```

Once again, we can ignore much of the assembly code and focus only on the important instructions of this function. As stated before, this binary runs strcmp to compare two strings. Here we can see the strcmp at the address `0x004006d5`. It will compare the string in the register `rdi` with the string in the register `rax`. If both strings are equal, it will set the 32bit `eax` to 0 and return the "password ok" message. The register `rax` holds the input we used as an argument, so what we need to know is the string that `rdi` holds before the function executes the strcmp. To do that, we set a breakpoint at the address of the instruction `mov rdi, rax` with the command `db 0x004006d2`. Then, we execute the program with `dc` until it reaches the breakpoint and stop. Since we need to provide an argument with the program, we can run `oop 'argument'`, and then set the breakpoint and run it (the string we use doesn't matter).

```

[0x7f6bd0d090]> ood 'random'
child received signal 9
Process with PID 3375 started ...
= attach 3375 3375
File dbg:///home/isildur/tryhackme/reversing/crackme4 'random' reopened in read-write mode
Unable to find filedescriptor 3
Unable to find filedescriptor 3
3375
[0x7fa79e13b090]> db 0x004006d2
[0x7fa79e13b090]> dc
hit breakpoint at: 4006d2
[0x004006d2]> pdf @sym.compare_pwd
; CALL XREF from main @ 0x400754
- 156: sym.compare_pwd (int64_t arg1);
    ; var int64_t var_28h @ rbp-0x28
    ; var int64_t var_20h @ rbp-0x20
    ; var int64_t var_18h @ rbp-0x18
    ; var int64_t var_10h @ rbp-0x10
    ; var int64_t var_eh @ rbp-0xe
    ; var int64_t var_8h @ rbp-0x8
    ; arg int64_t arg1 @ rdi
    0x0040067a      55          push rbp
    0x0040067b      4899e5      mov rbp, rsp
    0x0040067e      4883ec30  sub rsp, 0x30
    0x00400682      48897dd8  mov qword [var_28h], rdi    ; arg1
    0x00400686      64488b042528. mov rax, qword fs:[0x28]
    0x0040068f      488945f8  mov qword [var_8h], rax
    0x00400693      31c0        xor eax, eax
    0x00400695      48b8495d7b49. movabs rax, 0xb175614497b5d49
    0x0040069f      488945e0  mov qword [var_20h], rax
    0x004006a3      48b857414751. movabs rax, 0x547b175651474157
    0x004006ad      488945e8  mov qword [var_18h], rax
    0x004006b1      66c745f05340  mov word [var_10h], 0x4053 ; 'S'
    0x004006b7      c645f200  mov byte [var_eh], 0
    0x004006bb      488d45e0  lea rax, qword [var_20h]
    0x004006bf      4889c7    mov rdi, rax
    0x004006c2      e866ffff    call sym.imp.get_pwd
    0x004006c7      488b55d8  mov rdx, qword [var_28h]
    0x004006cb      488d45e0  lea rax, qword [var_20h]
    0x004006cf      4889d6    mov rsi, rdx
    ;-- rip:
    0x004006d2      b     4889c7    mov rdi, rax
    0x004006d5      e846feffff  call sym.imp.strcmp      ; int strcmp(const char *s1, const char *s2)
    0x004006da      85c0        test eax, eax
    0x004006dc      750c        jne 0x4006ea
    0x004006de      bfe8074000  mov edi, str.password_OK ; 0x4007e8 ; "password OK"
    0x004006e3      e8f8fdffff  call sym.imp.puts      ; int puts(const char *s)
    0x004006e8      eb16        jmp 0x400700
    0x004006ea      488b45d8  mov rax, qword [var_28h]
    0x004006ee      4889c6    mov rsi, rax
    0x004006f1      bff4074000  mov edi, str.password__s__not_OK ; 0x4007f4 ; "password \"%s\" not OK"
    0x004006f6      b800000000  mov eax, 0
    0x004006fb      e800feffff  call sym.imp.printf    ; int printf(const char *format)
    ; CODE XREF from sym.compare_pwd @ 0x4006e8
    > 0x00400700      488b45f8  mov rax, qword [var_8h]
    0x00400704      644833042528. xor rax, qword fs:[0x28]
    0x0040070d      7405        je 0x400714
    0x0040070f      e8dcfdffff  call sym.imp.__stack_chk_fail ; void __stack_chk_fail(void)

```

Now we've reached the breakpoint. To know the value of `rdi` we type `px @rdi`.

```
[0x004006d2]> px @rdi
- offset -
0x7ffc11aa39b0 6d79 5f6d 3072 335f 7365 6375 7233 5f70 my_m0r3_secuR3_p
0x7ffc11aa39c0 7764 0000 0000 0000 006b b743 9265 d413 we.....k.C.e..
0x7ffc11aa39d0 f039 aa11 fc7f 0000 5907 4000 0000 0000 .9 ... Y.aJ...
0x7ffc11aa39e0 d83a aa11 fc7f 0000 0000 0000 0200 0000 : ...
0x7ffc11aa39f0 6007 4000 0000 0000 0b2e f89d a77f 0000 ^.a.....
0x7ffc11aa3a00 0000 0000 0000 0000 d83a aa11 fc7f 0000 .....:.....
0x7ffc11aa3a10 0000 0400 0200 0000 1607 4000 0000 0000 .....@....
0x7ffc11aa3a20 0000 0000 0000 0000 3c92 6254 f31c 1fa0 .....<.bT...
0x7ffc11aa3a30 4005 4000 0000 0000 d03a aa11 fc7f 0000 @.a....:.....
0x7ffc11aa3a40 0000 0000 0000 0000 0000 0000 0000 0000 .....:.....
0x7ffc11aa3a50 3c92 a22e 273f e75f 3c92 2401 8327 505f <...?'._<.$..'_P_
0x7ffc11aa3a60 0000 0000 0000 0000 0000 0000 0000 0000 .....:.....
0x7ffc11aa3a70 0000 0000 0000 0000 f03a aa11 fc7f 0000 .....:.....
0x7ffc11aa3a80 9041 169e a77f 0000 6994 149e a77f 0000 .A....i.....
0x7ffc11aa3a90 0000 0000 0000 0000 0000 0000 0000 0000 .....:.....
0x7ffc11aa3aa0 4005 4000 0000 0000 d03a aa11 fc7f 0000 @.a....:.....
[0x004006d2]> 
```

We revealed the content of `rdi` before the `strcmp`, and we can check that it is indeed the correct password by running the binary with it as argument.

```
* ~ /tryhackme/reversing > ./crackme4
password OK
* ~ /tryhackme/reversing > 
```

## Crackme5

This binary asks for user input instead of arguments.

```
* ~ /tryhackme/reversing > ./crackme5
Enter your input:
hello
Always dig deeper
* ~ /tryhackme/reversing > 
```

Once again lets debug it with radare2 and disassemble the main function.

At the beginning of the function we see a different character assigned to each variable. Although quite troublesome, if we just assembled all this characters in a single string we would find out that the answer was always exposed to us.

```

0x0040078b    488945f8    mov qword [var_8h], rax
0x0040078f    31c0        xor eax, eax
0x00400791    c645d04f    mov byte [var_30h], 0x4f ; 'O' 79
0x00400795    c645d166    mov byte [var_2fh], 0x66 ; 'f' 102
0x00400799    c645d264    mov byte [var_2eh], 0x64 ; 'd' 100
0x0040079d    c645d36c    mov byte [var_2dh], 0x6c ; 'l' 108
0x004007a1    c645d444    mov byte [var_2ch], 0x44 ; 'D' 68
0x004007a5    c645d553    mov byte [var_2bh], 0x53 ; 'S' 83
0x004007a9    c645d641    mov byte [var_2ah], 0x41 ; 'A' 65
0x004007ad    c645d77c    mov byte [var_29h], 0x7c ; ']' 124
0x004007b1    c645d833    mov byte [var_28h], 0x33 ; '3' 51
0x004007b5    c645d974    mov byte [var_27h], 0x74 ; 't' 116
0x004007b9    c645da58    mov byte [var_26h], 0x58 ; 'X' 88
0x004007bd    c645db62    mov byte [var_25h], 0x62 ; 'b' 98
0x004007c1    c645dc33    mov byte [var_24h], 0x33 ; '3' 51
0x004007c5    c645dd32    mov byte [var_23h], 0x32 ; '2' 50
0x004007c9    c645de7e    mov byte [var_22h], 0x7e ; '~' 126
0x004007cd    c645df58    mov byte [var_21h], 0x58 ; 'X' 88
0x004007d1    c645e033    mov byte [var_20h], 0x33 ; '3' 51
0x004007d5    c645e174    mov byte [var_1fh], 0x74 ; 't' 116
0x004007d9    c645e258    mov byte [var_1eh], 0x58 ; 'X' 88
0x004007dd    c645e340    mov byte [var_1dh], 0x40 ; '@' 64
0x004007e1    c645e473    mov byte [var_1ch], 0x73 ; 's' 115
0x004007e5    c645e558    mov byte [var_1bh], 0x58 ; 'X' 88
0x004007e9    c645e660    mov byte [var_1ah], 0x60 ; ']' 96
0x004007ed    c645e734    mov byte [var_19h], 0x34 ; '4' 52
0x004007f1    c645e874    mov byte [var_18h], 0x74 ; 't' 116
0x004007f5    c645e958    mov byte [var_17h], 0x58 ; 'X' 88
0x004007f9    c645ea74    mov byte [var_16h], 0x74 ; 't' 116
0x004007fd    c645eb7a    mov byte [var_15h], 0x7a ; 'z' 122
0x00400801    bf54094000  mov edi, str.Enter_your_input; 0x400954 ; "Enter your input:"
0x00400806    e865fdffff  call sym.imp.puts ; int puts(const char *)
0x0040080b    488d45b0    lea rax, qword [var_50h]
0x0040080f    4889c6        mov rsi, rax
0x00400812    bf66094000  mov edi, 0x400966
0x00400817    b800000000  mov eax, 0
0x0040081c    e89ffdffff  call sym.imp.__isoc99_scanf ; int scanf(const char *format)
0x00400821    488d55d0    lea rdx, qword [var_30h]
0x00400825    488d45b0    lea rax, qword [var_50h]
0x00400829    4889d6        mov rsi, rdx
0x0040082c    4889c7        mov rdi, rax
0x0040082f    e8a2feffff  call symstrcmp ; int strcmp(const char *s1, const char *s2)
0x00400834    8945ac        mov dword [var_54h], eax
0x00400837    837dac00    cmp dword [var_54h], 0
0x0040083b    750c        jne 0x400849
0x0040083d    bf69094000  mov edi, str.Good_game ; 0x400969 ; "Good game"
0x00400842    e829fdffff  call sym.imp.puts ; int puts(const char *)
0x00400847    eb0a        jmp 0x400853
0x00400849    bf73094000  mov edi, str.Always_dig_deeper ; 0x400973 ; "Always dig deeper"
0x0040084e    e81dfdffff  call sym.imp.puts ; int puts(const char *)
; CODE XREF from main @ 0x400847
0x00400853    b800000000  mov eax, 0
0x00400858    488b4df8    mov rcx, qword [var_8h]
0x0040085c    6448330c2528. xor rcx, qword fs:[0x28]
0x00400865    7405        je 0x40086c
0x00400867    e824fdffff  call sym.imp.__stack_chk_fail ; void __stack_chk_fail(void)
0x0040086c    c9        leave
0x0040086d    c3        ret

```

We have once again a `strcmp`, this time in the main function. As before, we set a breakpoint before the `strcmp`, which compares the registers `rdi` and `rax`. This time, `rdi` doesn't store anything of value for us, but if we investigate the `rsi` register, we see that its content is a string with the same characters assigned to each variable at the beginning of the main function.

```
[0x7fa893691090]> dc
Enter your input:
password
hit breakpoint at: 40082c
[0x0040082c]> px @rdi
- offset -
  0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0xfffffa0d67f90 30a7 6b93 a87f 0000 8d34 feea 0000 0000 0.k...4...
0xfffffa0d67fa0 f0a4 6b93 a87f 0000 a303 4000 0000 0000 ..k...@...
0xfffffa0d67fb0 e880 d6a0 ff7f 0000 4080 d6a0 ff7f 0000 .....@...
0xfffffa0d67fc0 5080 d6a0 ff7f 0000 f1a3 6993 a87f 0000 P...i...
0xfffffa0d67fd0 0100 0000 0000 0000 a055 6793 a87f 0000 .....Ug...
0xfffffa0d67fe0 0100 0000 0000 0000 0000 0000 0000 0000 .....
0xfffffa0d67ff0 0100 0000 0000 0000 90a1 6b93 a87f 0000 .....k...
0xfffffa0d68000 0000 0000 0000 0000 a055 6793 a87f 0000 .....Ug...
0xfffffa0d68010 90a1 6b93 a87f 0000 f0a4 6b93 a87f 0000 ..k...k...
0xfffffa0d68020 0000 0000 0000 0000 0000 0000 0000 0000 .....
0xfffffa0d68030 0100 0000 0000 0000 0000 0000 0000 0000 .....
0xfffffa0d68040 ffff ffff 0000 0000 0000 0000 0000 0000 .....
0xfffffa0d68050 f8bb 4b93 a87f 0000 0050 6793 a87f 0000 ..K...Pg...
0xfffffa0d68060 2f2f 2f2f 2f2f 2f2f 2f2f 2f2f 2f2f 2f2f ///////////////
0xfffffa0d68070 0000 0000 0000 0000 0000 0000 0000 0000 .....
0xfffffa0d68080 0000 0000 0000 0000 0000 0000 0000 0000 .....
[0x0040082c]> px @rsi
- offset -
  0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0xfffffa0d68510 4f66 646c 4453 417c 3374 5862 3332 7e58 Ofd...X
0xfffffa0d68520 3374 5840 7358 6034 7458 747a 0000 0000 3tX...tz...
0xfffffa0d68530 2086 d6a0 ff7f 0000 006d 4c41 5809 1360 .....mLAX...
0xfffffa0d68540 d008 4000 0000 0000 0b8e 4d93 a87f 0000 ..@...M...
0xfffffa0d68550 0000 0000 0000 0000 2886 d6a0 ff7f 0000 .....(...
0xfffffa0d68560 0000 0400 0100 0000 7307 4000 0000 0000 .....s@...
0xfffffa0d68570 0000 0000 0000 0000 53d9 5cf0 5638 b49c .....S.\V8...
0xfffffa0d68580 e005 4000 0000 0000 2086 d6a0 ff7f 0000 ..@.....
0xfffffa0d68590 0000 0000 0000 0000 0000 0000 0000 0000 .....
0xfffffa0d685a0 53d9 5ce7 7b79 4b63 53d9 7af6 4d1e e563 S.\{yKcS.z.M..c
0xfffffa0d685b0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0xfffffa0d685c0 0000 0000 0000 0000 3886 d6a0 ff7f 0000 .....8...
0xfffffa0d685d0 90a1 6b93 a87f 0000 69f4 6993 a87f 0000 ..k...i.i...
0xfffffa0d685e0 0000 0000 0000 0000 0000 0000 0000 0000 .....
0xfffffa0d685f0 e005 4000 0000 0000 2086 d6a0 ff7f 0000 ..@.....
0xfffffa0d68600 0000 0000 0000 0000 0906 4000 0000 0000 .....@...
[0x0040082c]>
```

If we use this string as the input we get the "Good game" message.

```
x ~ /tryhackme/reversing ./crackme5
Enter your input:
0
Good game
~ /tryhackme/reversing
```

## Crackme6

Crackme6, like other previous crackmes, asks for a password as argument.

```
~ /tryhackme/reversing ./crackme6
Usage : ./crackme6 password
Good luck, read the source
~ /tryhackme/reversing ./crackme6 password
password "password" not OK
~ /tryhackme/reversing
```

Let's analyze the main function.

```
① ~ /tryhackme/reversing ➤ r2 -d crackme6
Process with PID 4583 started ...
= attach 4583 4583
bin.baddr 0x00400000
Using 0x400000
asm.bits 64
Warning: r_bin_file_hash: file exceeds bin.hashlimit
[0x7ff7a9d44090]> aaa
[x] Analyze all flags starting with sym. and entry0 (aa)
[x] Analyze function calls (aac)
[x] Analyze len bytes of instructions for references (aar)
[x] Check for objc references
[x] Check for vtables
[TOFIX: aaf can't run in debugger mode.ions (aaf)]
[x] Type matching analysis for all functions (aaf)
[x] Propagate noreturn information
[x] Use -AA or aaaa to perform additional experimental analysis.
[0x7ff7a9d44090]> pdf @main
    ; DATA XREF from entry0 @ 0x4004ad
74: int main (int argc, char **argv, char **envp);
    ; var int64_t var_10h @ rbp-0x10
    ; var int64_t var_4h @ rbp-0x4
    ; arg int argc @ rdi
    ; arg char **argv @ rsi
    0x00400711      55          push rbp
    0x00400712      4889e5      mov rbp, rsp
    0x00400715      4883ec10   sub rsp, 0x10
    0x00400719      897dfc      mov dword [var_4h], edi ; argc
    0x0040071c      488975f0   mov qword [var_10h], rsi ; argv
    0x00400720      837dfc02   cmp dword [var_4h], 2
    < 0x00400724      741b       je 0x400741
    0x00400726      488b45f0   mov rax, qword [var_10h]
    0x0040072a      488b00       mov rax, qword [rax]
    0x0040072d      4889c6       mov rsi, rax
    0x00400730      bf10084000  mov edi, str.Usage:_s_password_Good_luck_read_the_source ; 0:
; "Usage : %s password\nGood luck, read the source\n"
    0x00400735      b800000000  mov eax, 0
    0x0040073a      e821fdffff  call sym.imp.printf ; int printf(const char *format)
    < 0x0040073f      eb13       jmp 0x400754
    > 0x00400741      488b45f0   mov rax, qword [var_10h]
    0x00400745      4883c008   add rax, 8
    0x00400749      488b00       mov rax, qword [rax]
    0x0040074c      4889c7       mov rdi, rax
    0x0040074f      e87dffffff  call sym.compare_pwd
; CODE XREF from main @ 0x40073f
    > 0x00400754      b800000000  mov eax, 0
    0x00400759      c9          leave
    0x0040075a      c3          ret
[0x7ff7a9d44090]> []
```

Once again let's focus on the essential instructions of the function. What caught my attention here is the `sym.compare_pwd`, so let's analyze it.

```
[0x7ff7a9d44090]> pdf @sym.compare_pwd
      ; CALL XREF from main @ 0x40074f
64: sym.compare_pwd (int64_t arg1);
    ; var int64_t var_8h @ rbp-0x8
    ; arg int64_t arg1 @ rdi
    0x004006d1      55          push rbp
    0x004006d2      4889e5      mov rbp, rsp
    0x004006d5      4883ec10    sub rsp, 0x10
    0x004006d9      48897df8    mov qword [var_8h], rdi      ; arg1
    0x004006dd      488b45f8    mov rax, qword [var_8h]
    0x004006e1      4889c7      mov rdi, rax
    0x004006e4      e894feffff  call sym.my_secure_test
    0x004006e9      85c0        test eax, eax
    0x004006eb      750c        jne 0x4006f9
    0x004006ed      bfe8074000  mov edi, str.password_OK ; 0x4007e8 ; "password OK"
    0x004006f2      e859fdffff  call sym.imp.puts       ; int puts(const char *s)
    < 0x004006f7      eb16        jmp 0x40070f
    < 0x004006f9      488b45f8    mov rax, qword [var_8h]
    < 0x004006fd      4889c6      mov rsi, rax
    < 0x00400700      bff4074000  mov edi, str.password__s_not_OK ; 0x4007f4 ; "password \"%s\" not OK\\"
    > 0x00400705      b800000000  mov eax, 0
    > 0x0040070a      e851fdffff  call sym.imp.printf    ; int printf(const char *format)
    ; CODE XREF from sym.compare_pwd @ 0x4006f7
    > 0x0040070f      c9          leave
    > 0x00400710      c3          ret
[0x7ff7a9d44090]>
```

We can see that it is in this function that the password is checked with our input, but we cannot know what our input is actually being compared to because that value comes from the `sym.my_secure_test` function. So let's take a look at it.

```

0x004005bc    0fb600      movzx eax, byte [rax]
0x004005bf    3c33        cmp al, 0x33           ; 51
< 0x004005c1    740a        je 0x4005cd
    < 0x004005c3    b8fffffff
    < 0x004005c8    e902010000   mov eax, 0xffffffff ; -1
    > 0x004005cd    488b45f8   jmp 0x4006cf
0x004005d1    4883c002   mov rax, qword [var_8h]
0x004005d5    0fb600      add rax, 2
0x004005d8    84c0        movzx eax, byte [rax]
0x004005da    740f        test al, al
< 0x004005dc    488b45f8   je 0x4005eb
0x004005e0    4883c002   mov rax, qword [var_8h]
0x004005e4    0fb600      add rax, 2
0x004005e7    3c33        movzx eax, byte [rax]
< 0x004005e9    740a        cmp al, 0x33           ; 51
    < 0x004005eb    b8fffffff
    < 0x004005f0    e9da000000   mov eax, 0xffffffff ; -1
    > 0x004005f5    488b45f8   jmp 0x4006cf
0x004005f9    4883c003   mov rax, qword [var_8h]
0x004005fd    0fb600      add rax, 3
0x00400600    84c0        movzx eax, byte [rax]
0x00400602    740f        test al, al
< 0x00400604    488b45f8   je 0x400613
0x00400608    4883c003   mov rax, qword [var_8h]
0x0040060c    0fb600      add rax, 3
0x0040060f    3c37        movzx eax, byte [rax]
0x00400611    740a        cmp al, 0x37           ; 55
< 0x00400613    b8fffffff
    < 0x00400618    e9b2000000   mov eax, 0xffffffff ; -1
    > 0x0040061d    488b45f8   jmp 0x4006cf
0x00400621    4883c004   mov rax, qword [var_8h]
0x00400625    0fb600      add rax, 4
0x00400628    84c0        movzx eax, byte [rax]
< 0x0040062a    740f        test al, al
    < 0x0040062c    488b45f8   je 0x40063b
0x00400630    4883c004   mov rax, qword [var_8h]
0x00400634    0fb600      add rax, 4
0x00400637    3c5f        movzx eax, byte [rax]
< 0x00400639    740a        cmp al, 0x5f           ; 95
    < 0x0040063b    b8fffffff
    < 0x00400640    e98a000000   mov eax, 0xffffffff ; -1
    > 0x00400645    488b45f8   jmp 0x4006cf
0x00400649    4883c005   mov rax, qword [var_8h]
0x0040064d    0fb600      add rax, 5
0x00400650    84c0        movzx eax, byte [rax]
0x00400652    740f        test al, al
< 0x00400654    488b45f8   je 0x400663
0x00400658    4883c005   mov rax, qword [var_8h]
0x0040065c    0fb600      add rax, 5
0x0040065f    3c70        movzx eax, byte [rax]
< 0x00400661    7407        cmp al, 0x70           ; 112
    < 0x00400663    b8fffffff
    < 0x00400668    eb65        mov eax, 0xffffffff ; -1
    > 0x0040066a    488b45f8   jmp 0x4006cf
0x0040066e    4883c006   mov rax, qword [var_8h]
0x00400672    0fb600      add rax, 6
0x00400675    84c0        movzx eax, byte [rax]
0x00400677    740f        test al, al
< 0x00400679    488b45f8   je 0x400688
0x0040067d    4883c006   mov rax, qword [var_8h]
0x0040067d    add rax, 6

```

Ok! This function is too big for a single screenshot, and its really hard to analyze with all those jumps, specially in text mode. This where graph mode comes in handy. To enter graph mode just type `vV`. Once in graph mode we can use the same commands as in text mode, but we need to start them with `:` instead.



Graph mode makes things more easy to understand. We can see that this function compares the least significant 8 bit of the `rax` register ( `al` ) with some hex value, then if its equal it increments the `rax` register and compares it again with another hex value. It seems that it is comparing every char in our input with these hex values.

```
0x400590 [ob]
mov rax, qword [var_8h]
movzx eax, byte [rax]
; 49
cmp al, 0x31
je 0x4005a5
```

```
0x4005b4 [oe]
mov rax, qword [var_8h]
add rax, 1
movzx eax, byte [rax]
; 51
cmp al, 0x33
je 0x4005cd
```

```
0x4005dc [oh]
mov rax, qword [var_8h]
add rax, 2
movzx eax, byte [rax]
; 51
cmp al, 0x33
je 0x4005f5
```

```
0x400604 [ok]
mov rax, qword [var_8h]
add rax, 3
movzx eax, byte [rax]
; 55
cmp al, 0x37
je 0x40061d
```

```
0x40062c [on]
mov rax, qword [var_8h]
add rax, 4
movzx eax, byte [rax]
; 95
cmp al, 0x5f
je 0x400645
```

```
0x400654 [oq]
mov rax, qword [var_8h]
add rax, 5
movzx eax, byte [rax]
; 112
cmp al, 0x70
je 0x40066a
```

```
0x400679 [ot]
mov rax, qword [var_8h]
add rax, 6
movzx eax, byte [rax]
; 119
cmp al, 0x77
je 0x40068f
```

```
0x40069e [ow]
mov rax, qword [var_8h]
add rax, 7
movzx eax, byte [rax]
; 100
cmp al, 0x64
je 0x4006b4
```

If we join all these values together and decode them into their ASCII equivalent we get the password to solve the crackme.

```
⌚ ➤ ~/tryhackme/reversing ➤ python3
Python 3.8.3rc1 (default, Apr 30 2020, 07:33:30)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> bytes.fromhex('313333375f707764').decode('utf8')
'1333375f707764'
>>>
```

We can confirm it is the right password by running the binary.

```
✗ ⌚ ➤ ~/tryhackme/reversing ➤ ./crackme6 : 
password OK
⌚ ➤ ~/tryhackme/reversing ➤
```

## Crackme7

When we execute this binary it has a basic interactive console where we can say hello and add numbers.













