TryHackMe Room Write-up

Title: Reversing ELF

Description: Room for beginner Reverse Engineering CTF players.

Author: Igor Buszta

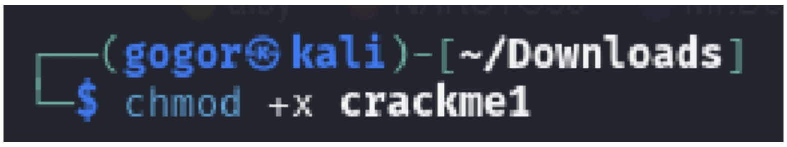
Crackme1

Description: *Let's start with a basic warmup, can you run the binary?*

***Objective**: What is the flag?*

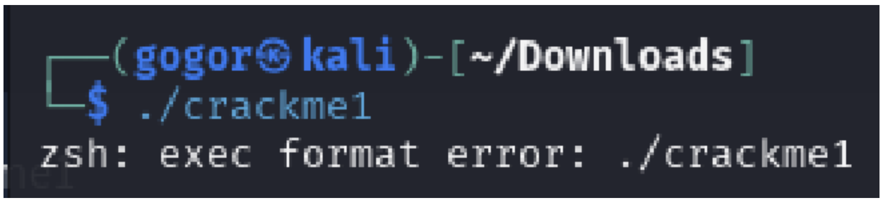
***Tools:** Kali Linux on UTM (macOS M1)*

The objective is to basically run the file. Let's add proper permissions and do ./crackme1



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

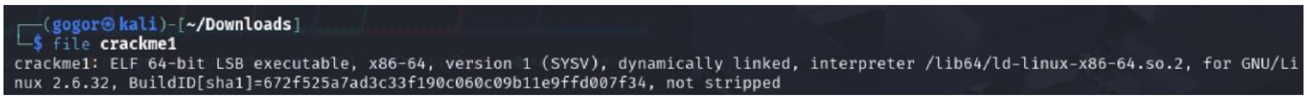
Picture 1: Adding 'executable' permission to the file.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ ./crackme1
zsh: exec format error: ./crackme1
```

Picture 2: Running the file using ./

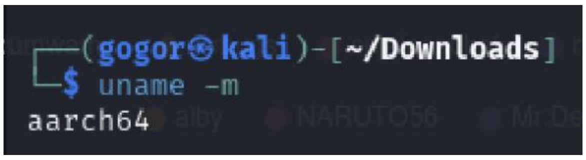
Okay, so the system screams that there's a problem with our file format. Let's take a look what is this file anyway.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ file crackme1
crackme1: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Li
nux 2.6.32, BuildID[sha1]=672f525a7ad3c33f190c060c09b11e9ffd007f34, not stripped
```

Picture 3: File type informations.

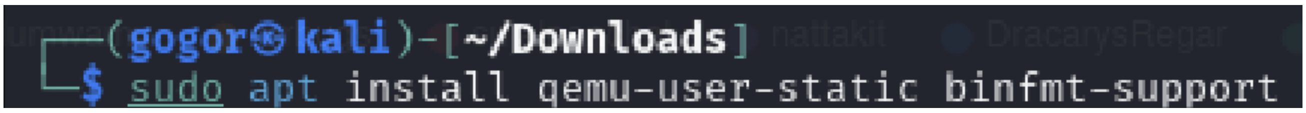
The file is 64-bit made in x86-64 architecture. Knowing my device, it might be different. Let's use *uname -m* to see the architechture we're working on right now.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ uname -m
aarch64
```

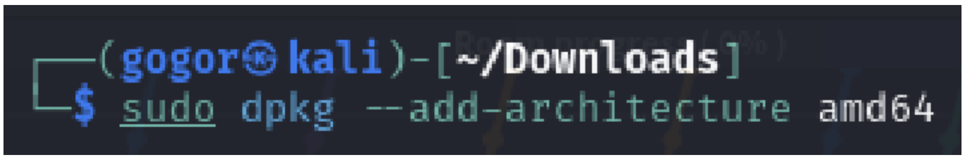
Picture 4: Device's architecture.

Okay, so we need to install proper emulators and add architectures. Let's install and build them.



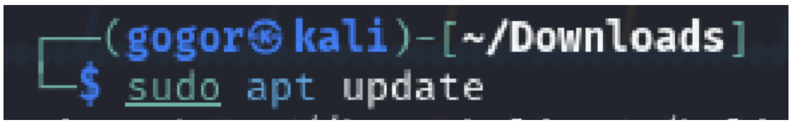
```
┌──(gogor㉿kali)-[~/Downloads]
└─$ sudo apt install qemu-user-static binfmt-support
```

Picture 5: Installing QEMU emulator.



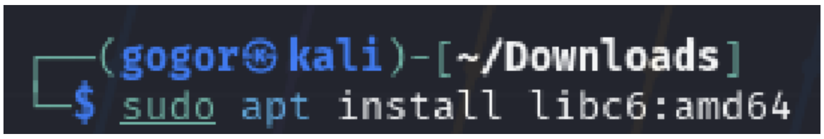
```
┌──(gogor㉿kali)-[~/Downloads]
└─$ sudo dpkg --add-architecture amd64
```

Picture 6: Adding amd64 architecture into the system.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ sudo apt update
```

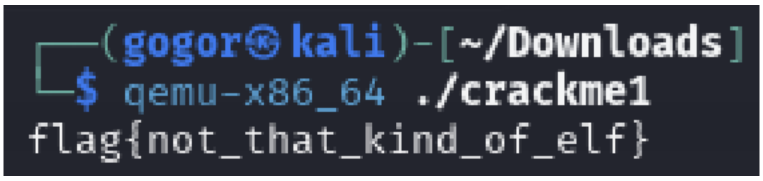
Picture 7: Updating all packages in the system.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ sudo apt install libc6:amd64
```

Picture 8: Installing required libraries in order to execute the file.

We're all set. Let's now run the file on emualted architecture.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ qemu-x86_64 ./crackme1
flag{not_that_kind_of_elf}
```

Picture 9: Getting our first flag.

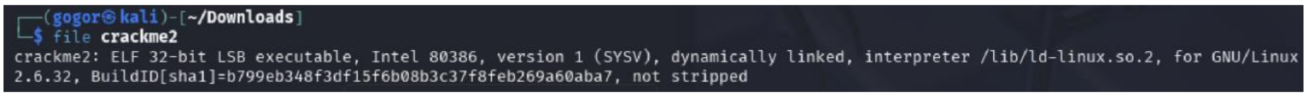
Crackme2

Description: *Find the super-secret password! and use it to obtain the flag*

***Objective**: What is the super secret password? What is the flag?*

***Tools:** Kali Linux on UTM (macOS M1), strings*

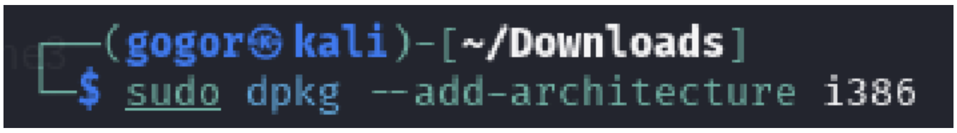
Knowing what we know from previous task, let’s first check the type of the file.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ file crackme2
crackme2: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux
2.6.32, BuildID[sha1]=b799eb348f3df15f6b08b3c37f8feb269a60aba7, not stripped
```

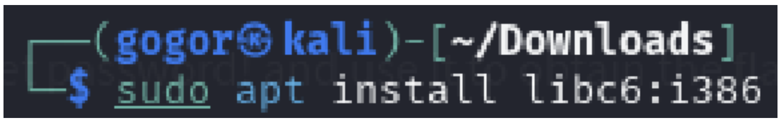
Picture 10: Information of crackme2 file.

We’re onto something. This time the file is 32-bit in architecture of Intel 80386 (i386). Once again we need to get proper things installed in order to run this file.



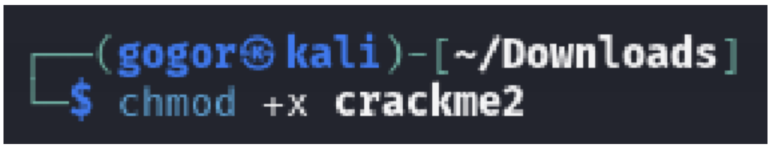
```
┌──(gogor㉿kali)-[~/Downloads]
└─$ sudo dpkg --add-architecture i386
```

Picture 11: Adding the i386 architecture in the system.



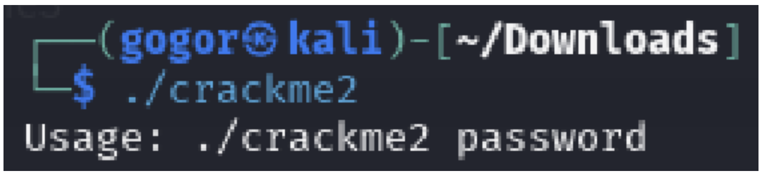
```
┌──(gogor㉿kali)-[~/Downloads]
└─$ sudo apt install libc6:i386
```

Picture 12: Installing the standard library for the architecture.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ chmod +x crackme2
```

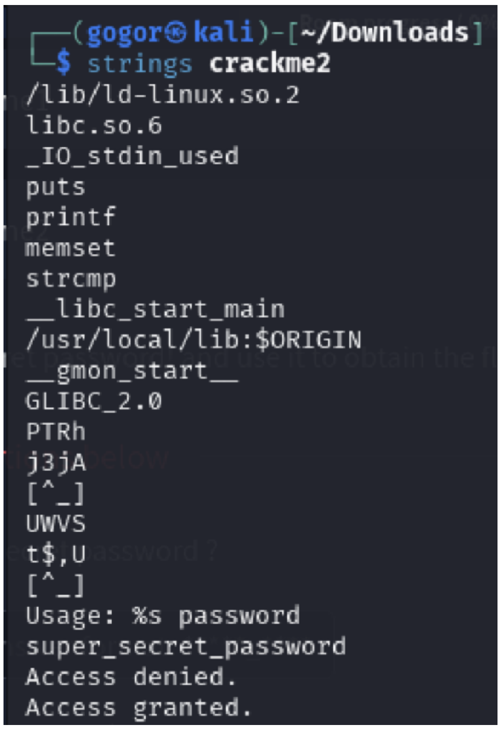
Picture 13: Adding the ‘executable’ permission to the file.



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

Picture 14: Running the file.

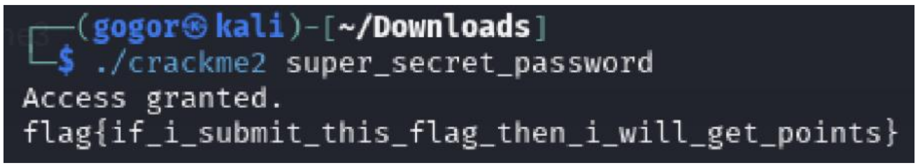
So we have some code behind this file this time. Let's peek inside to see if there are any strings hidden withing the file using *strings* tool.



```
┌──(gogor㉿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
Access denied.
Access granted.
```

Picture 15: Fragment of strings output.

As we can see, unless the password is *super_secret_password* we will get message „Access denied". So, let's type it in.



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

Picture 16: Entered password and results.

We've found our second flag.

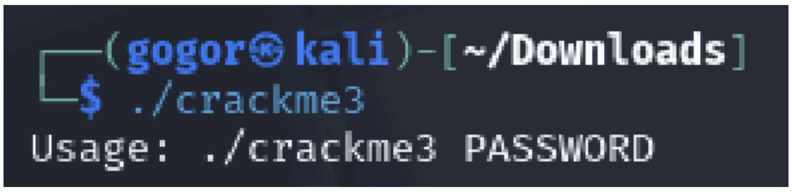
Crackme3

Description: *Use basic reverse engineering skills to obtain the flag*

***Objective**: What is the flag?*

***Tools:** Kali Linux on UTM (macOS M1), strings, online decoder*

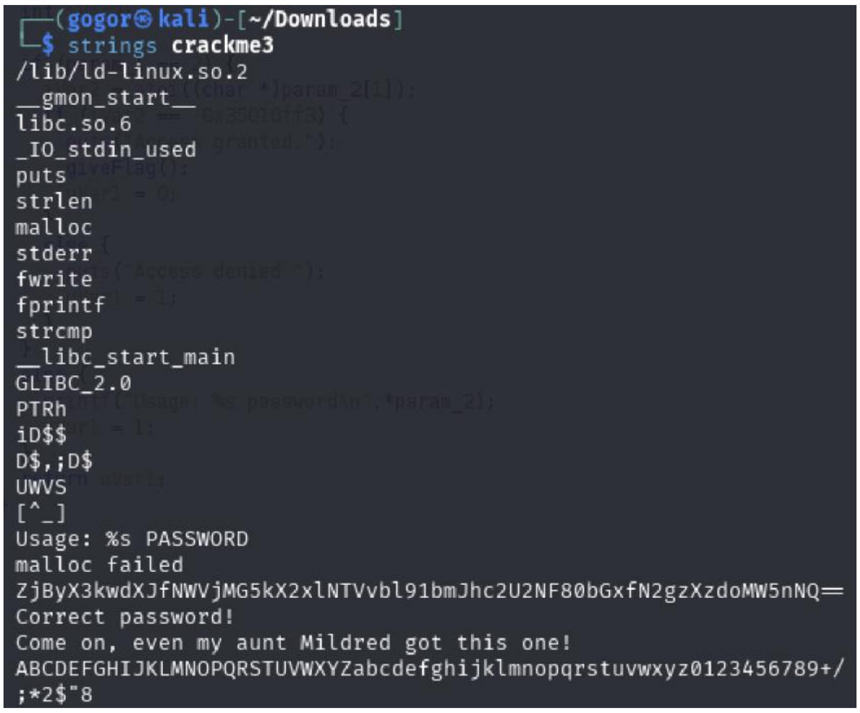
Now that we have all the libraries and an emulator, we can head straight into analizing the files and giving them permissions (chmod +x crackme[Y] will be omitted further in the write-up).



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ ./crackme3
Usage: ./crackme3 PASSWORD
```

Picture 17: Usage of crackme3 command.

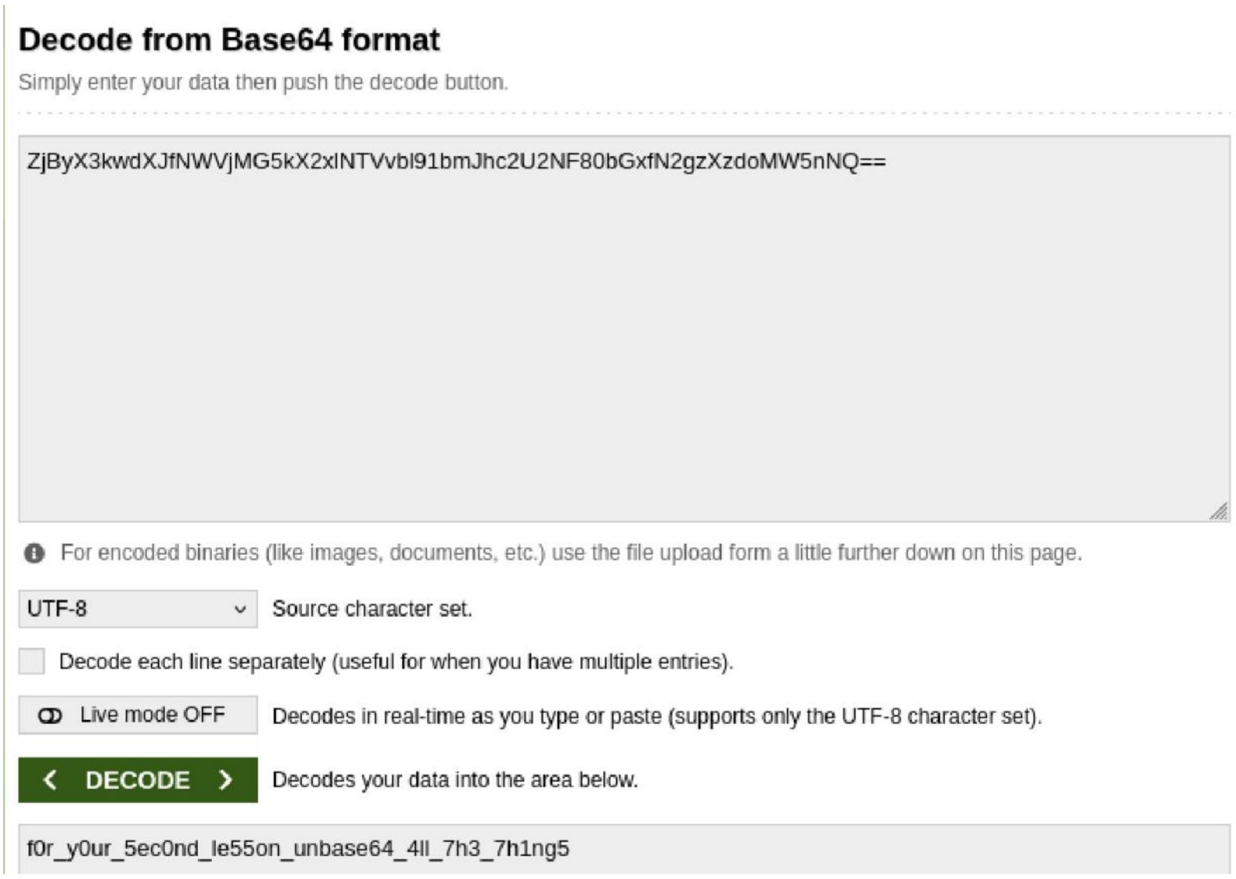
We have to find another password then. Let's hope that strings will once again come in handy.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ 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
ZjByX3kwdXJfNWVjMG5kX2xlNTVvbl91bmJhc2U2NF80bGxfN2gzXzdoMW5nNQ==
Correct password!
Come on, even my aunt Mildred got this one!
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
;*2$"8
```

Picture 18: Output of strings command used on crackme3.

Looks like there's some encoded message in there. It ends with '==' which indicates base64 encoding. Let's use first decoder found in the internet.


Decode from Base64 format

Simply enter your data then push the decode button.

```
ZjByX3kwdXJfNWVjMG5kX2xlNTVvbl91bmJhc2U2NF80bGxfN2gzXzdoMW5nNQ==
```

For encoded binaries (like images, documents, etc.) use the file upload form a little further down on this page.

UTF-8 — Source character set.

Decode each line separately (useful for when you have multiple entries).

Live mode OFF — Decodes in real-time as you type or paste (supports only the UTF-8 character set).

DECODE — Decodes your data into the area below.

```
f0r_y0ur_5ec0nd_le55on_unbase64_4ll_7h3_7h1ng5
```

Picture 19: Decoding the message found in file.

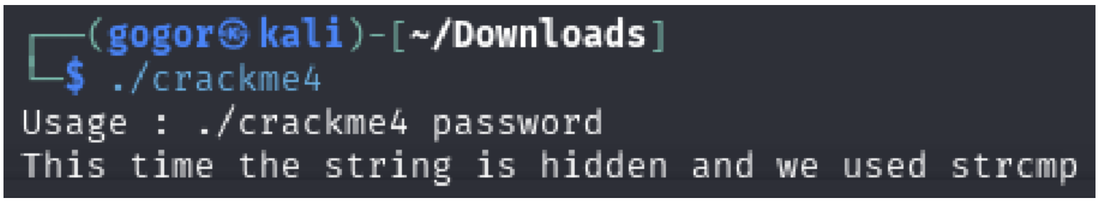
Crackme4

Description: *Analyze and find the password for the binary?*

***Objective**: What is the password?*

***Tools:** Kali Linux on UTM (macOS M1), strings, Ghidra, Python*

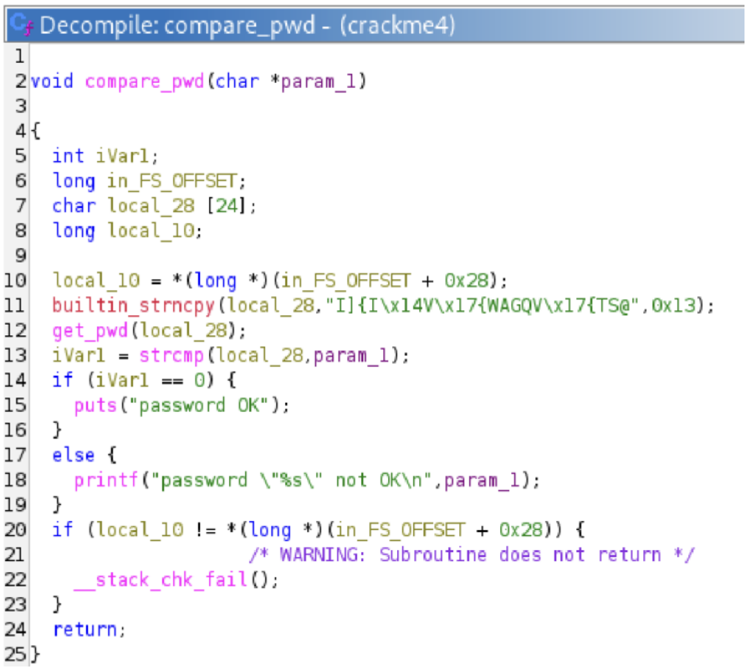
Once again let's add permissions and run the file.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ ./crackme4
Usage : ./crackme4 password
This time the string is hidden and we used strcmp
```

Picture 20: Running the crackme4 file.

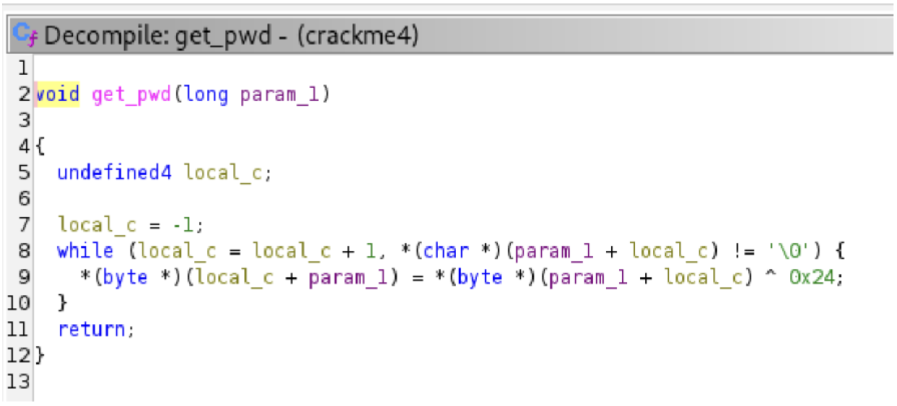
Now we are informed that getting this flag won't be so easy. They used function strcmp() which compares strings and returns 0 if they're the same. The string is most likely compared with given argument, but where to find it? We need to reverse engineer the code (finally). I used Ghidra since I'm already familiar with it. (First use in Compiled room).



Decompile: compare_pwd - (crackme4)

```c

void compare_pwd(char *param_1)

{
  int iVar1;
  long in_FS_OFFSET;
  char local_28 [24];
  long local_10;

  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  builtin_strncpy(local_28,"I]{I\x14V\x17{WAGQV\x17{TS@",0x13);
  get_pwd(local_28);
  iVar1 = strcmp(local_28,param_1);
  if (iVar1 == 0) {
    puts("password OK");
  }
  else {
    printf("password \"%s\" not OK\n",param_1);
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}
```

Picture 21: compare_pwd function – Ghidra.



Decompile: get_pwd - (crackme4)

```c

void get_pwd(long param_1)

{
  undefined4 local_c;

  local_c = -1;
  while (local_c = local_c + 1, *(char *)(param_1 + local_c) != '\0') {
    *(byte *)(local_c + param_1) = *(byte *)(param_1 + local_c) ^ 0x24;
  }
  return;
}

```

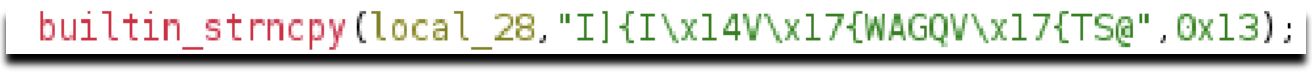
Picture 22: get_pwd function – Ghidra.

Okay, let's break it down. The main() function (not included in the picture) uses both compare_pwd() and get_pwd() functions.

In comapare_pwd() the code compares variable local_28 and param_1 (argument that we input).

What is local_28 then?

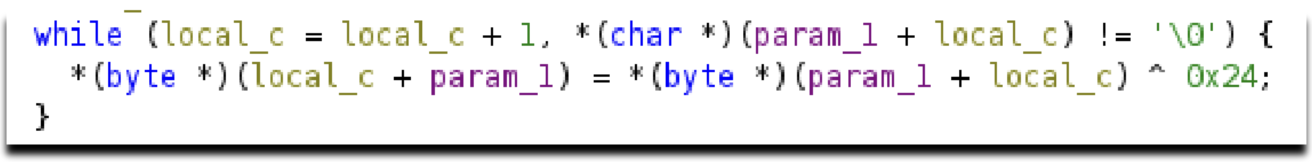
We can see that there's a string copied into the local_28 by builtin_strncpy and then passed to get_pwd().



```
builtin_strncpy(local_28,"I]{I\x14V\x17{WAGQV\x17{TS@",0x13);
```

Picture 23: Given string being copied into local_28 variable.

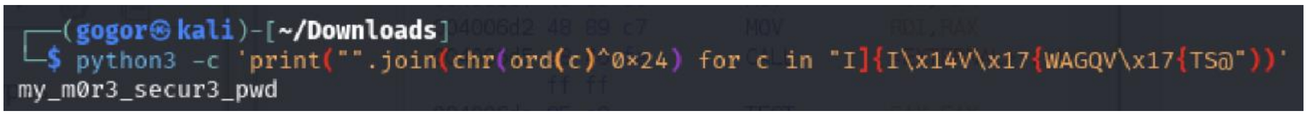
get_pwd() then goes through every character and does XOR operation with 0x24.



```
while (local_c = local_c + 1, *(char *)(param_1 + local_c) != '\0') {
  *(byte *)(local_c + param_1) = *(byte *)(param_1 + local_c) ^ 0x24;
}
```

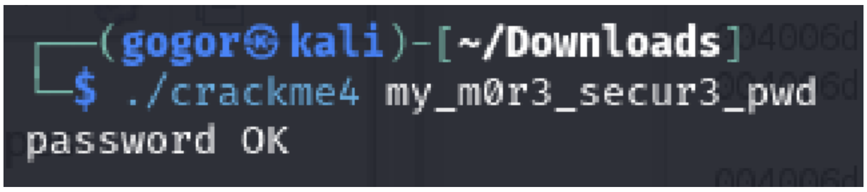
Picture 24: while loop going through every character (until the '\0' sign) and XORing (^) them by 0x24

All we need to do then is to reverse that XOR operation. I used simple Python one liner to do that.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ python3 -c 'print("".join(chr(ord(c)^0x24) for c in "I]{I\x14V\x17{WAGQV\x17{TS@"))'
my_m0r3_secur3_pwd
```

Picture 25: Reversing XOR operation and getting the password.



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

Picture 26: Checking if the password is correct.

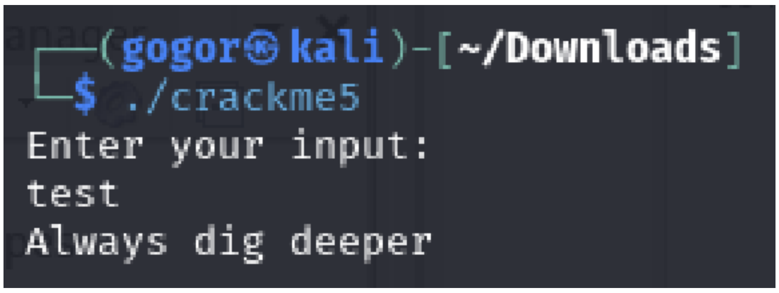
Crackme5

Description: *What will be the input of the file to get output Good game ?*

***Objective**: What is the input ?*

***Tools:** Kali Linux on UTM (macOS M1), Ghidra, Python*

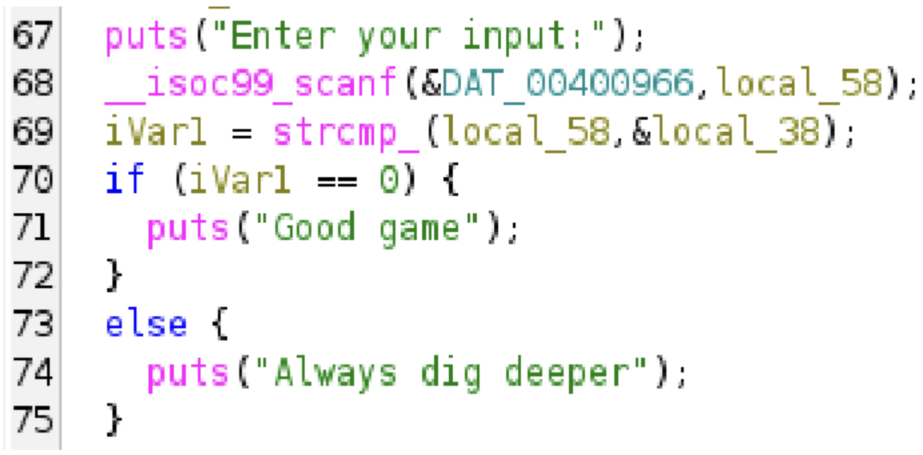
Let's check how the crackme5 functions.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ ./crackme5
Enter your input:
test
Always dig deeper
```

Picture 27: Test-run of crackme5 file.

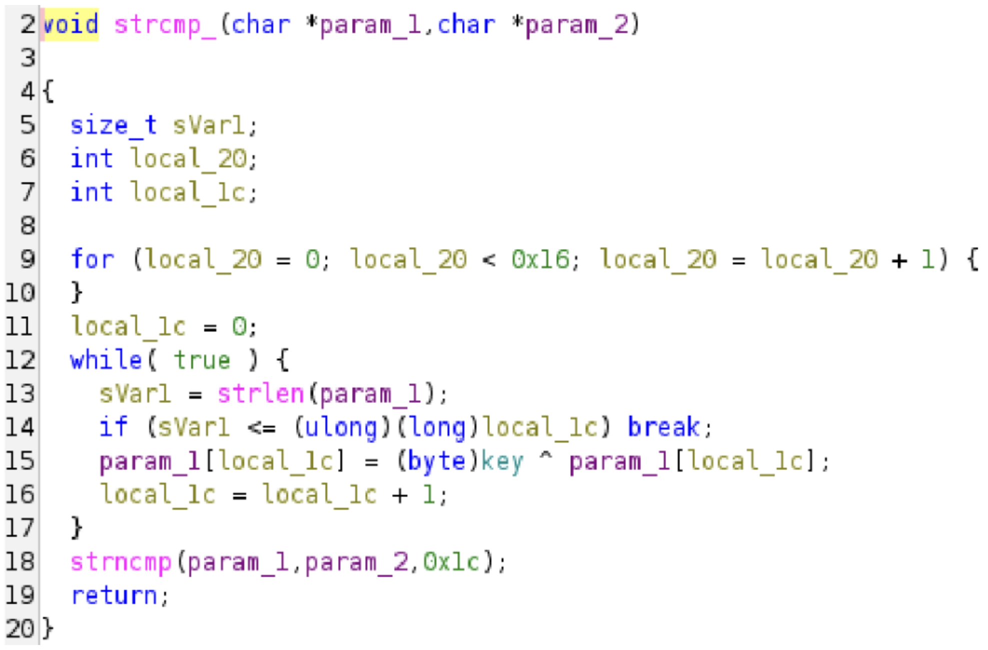
The file requires input from the user, compares it to something and gives an anwser based on the result of the comparison. Let's not waste time and head straight into Ghidra and look at the code.



```
puts("Enter your input:");
__isoc99_scanf(&DAT_00400966,local_58);
iVar1 = strcmp_(local_58,&local_38);
if (iVar1 == 0) {
  puts("Good game");
}
else {
  puts("Always dig deeper");
}
```

Picture 28: Part of the main() function – Ghidra.

There's basic logic there. But there's a strcmp_() function which is not built-in. Let's see what's going on in there.

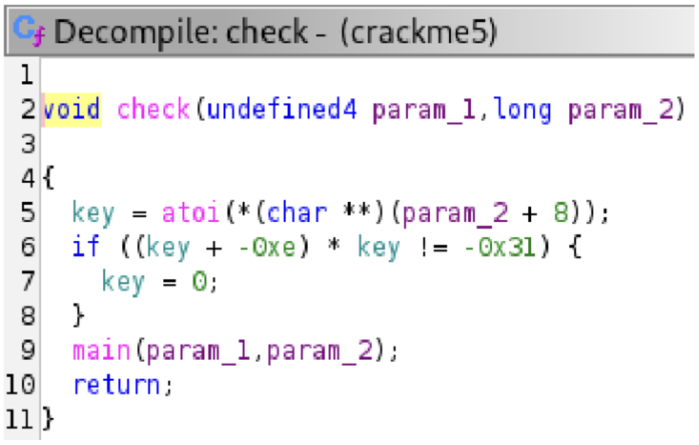


```
void strcmp_(char *param_1,char *param_2)

{
  size_t sVar1;
  int local_20;
  int local_1c;

  for (local_20 = 0; local_20 < 0x16; local_20 = local_20 + 1) {
  }
  local_1c = 0;
  while( true ) {
    sVar1 = strlen(param_1);
    if (sVar1 <= (ulong)(long)local_1c) break;
    param_1[local_1c] = (byte)key ^ param_1[local_1c];
    local_1c = local_1c + 1;
  }
  strncmp(param_1,param_2,0x1c);
  return;
}
```

Picture 29: The strcmp_() function – Ghidra.

Let’s dive in. The for loop has nothing between brackets, therefore it does nothing.

More interesting is the while loop. It takes param_1 and XORs it with some key, character by character. Let’s find and see where is this key defined.



```
void check(undefined4 param_1,long param_2)

{
  key = atoi(*(char **)(param_2 + 8));
  if ((key + -0xe) * key != -0x31) {
    key = 0;
  }
  main(param_1,param_2);
  return;
}
```

Picture 30: key variable being defined

The key is the first argument inputed by user (param_2 + 8 is equal to argv[1]) being transformed into integer using atoi(). What’s interesting is that the key will always be 0 unless the condition is true. To simplify the if conditio let’s transform HEX into DEC.

$$(key\ + (-14)) * key\ = \ -49$$

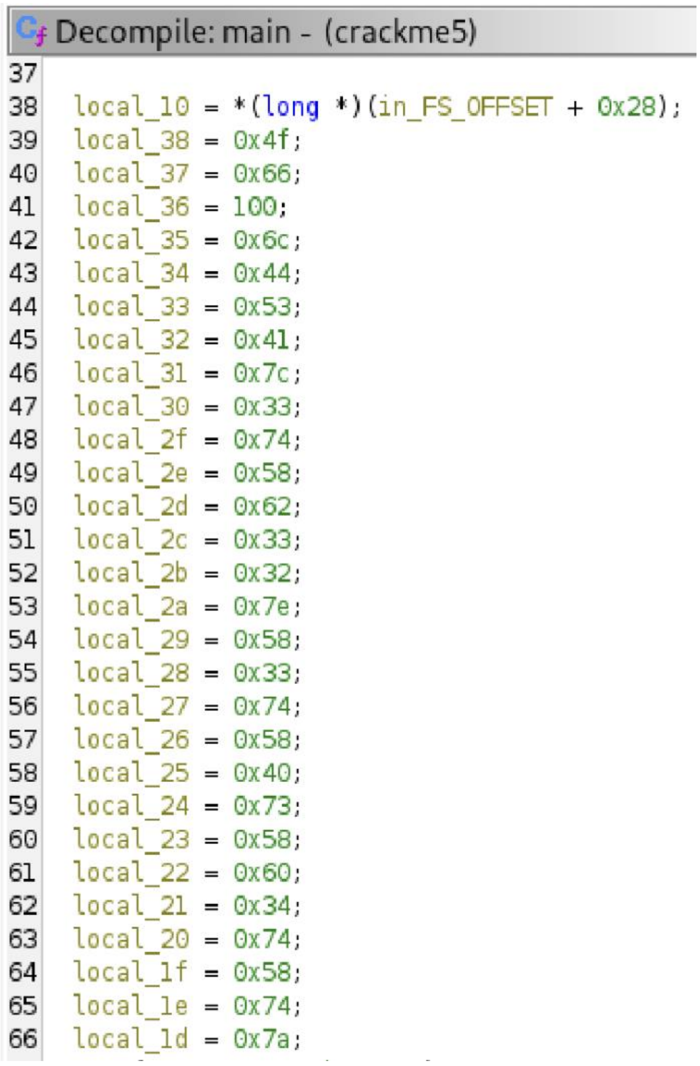
$$key^2 - 14key + 49 = 0$$

Say what now? We're solving quadriatic formulas now? I guess so. Luckly no deltas are required because it's simple multiplication formula.

$$(key - 7)^2 = 0$$

So now our key has to be 7 otherwise it will always be 0.

After going back to strcmp_() we now know that some string is being XORed with 7.

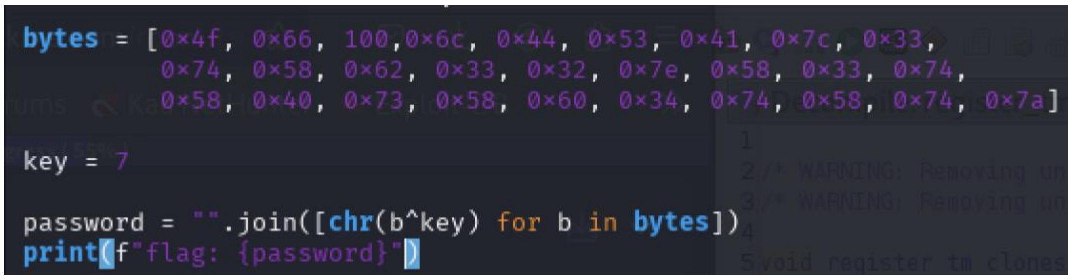

Decompile: main - (crackme5)

```
  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_38 = 0x4f;
  local_37 = 0x66;
  local_36 = 100;
  local_35 = 0x6c;
  local_34 = 0x44;
  local_33 = 0x53;
  local_32 = 0x41;
  local_31 = 0x7c;
  local_30 = 0x33;
  local_2f = 0x74;
  local_2e = 0x58;
  local_2d = 0x62;
  local_2c = 0x33;
  local_2b = 0x32;
  local_2a = 0x7e;
  local_29 = 0x58;
  local_28 = 0x33;
  local_27 = 0x74;
  local_26 = 0x58;
  local_25 = 0x40;
  local_24 = 0x73;
  local_23 = 0x58;
  local_22 = 0x60;
  local_21 = 0x34;
  local_20 = 0x74;
  local_1f = 0x58;
  local_1e = 0x74;
  local_1d = 0x7a;
```

Picture 31: Variabled representing ints side by side in program's memory.

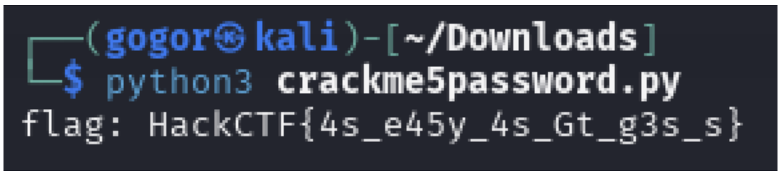
Why are there so many local variables and what do they mean? They were most likely one table in original code but Ghidra doesn't know that, and creates seperate variables instead of one table. What's important is that they represent continuos data in the memory (pointers are used as arguments in functions). That's our string which is being XORed with 7! Notice that &local_38 (address of the first character in string) is one of the arguments in strcmp_() (the other is our input).

After all the analyzing, let's get to coding. We need a script which takes each variable (their HEX value) and XOR sit with 7. One liner won't be enough, because we need to retype the values into our Python script.



```
bytes = [0x4f, 0x66, 100,0x6c, 0x44, 0x53, 0x41, 0x7c, 0x33,
         0x74, 0x58, 0x62, 0x33, 0x32, 0x7e, 0x58, 0x33, 0x74,
         0x58, 0x40, 0x73, 0x58, 0x60, 0x34, 0x74, 0x58, 0x74, 0x7a]

key = 7

password = "".join([chr(b^key) for b in bytes])
print(f"flag: {password}")
```

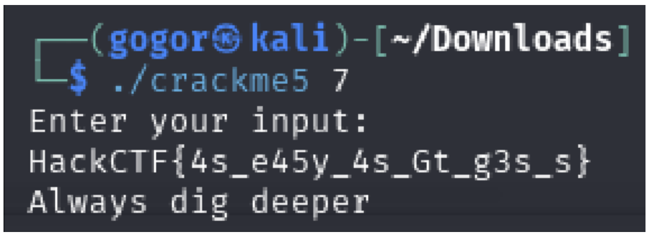
Picture 32: Python script which reverses the XOR operation.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ python3 crackme5password.py
flag: HackCTF{4s_e45y_4s_Gt_g3s_s}
```

Picture 33: Output of the script.

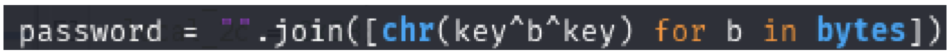
We now have some flag, but it doesn't fit the format and doesn't gives us the expected output.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ ./crackme5 7
Enter your input:
HackCTF{4s_e45y_4s_Gt_g3s_s}
Always dig deeper
```

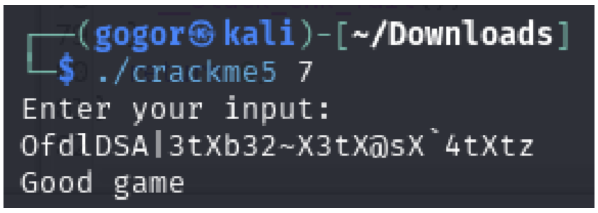
Picture 34: Checking the input and if flag is correct.

After messing and looking for anwser, I just XORed this plain flag once again by 7 to input the gibberish and hope, that comparison implemented in the code would work.



```
password = "".join([chr(key^b^key) for b in bytes])
```

Picture 35: Change made in Python script.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ ./crackme5 7
Enter your input:
OfdlDSA|3tXb32~X3tX@sX`4tXtz
Good game
```

Picture 36: Using the script's output as input in crackme5.

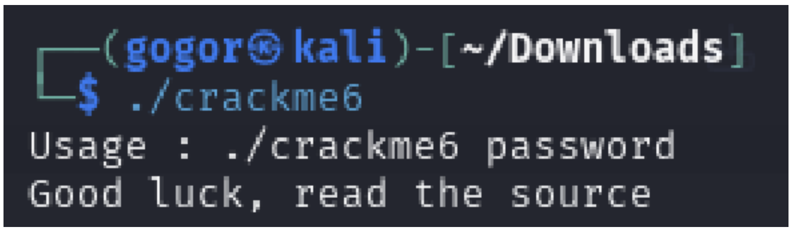
Crackme6

Description: *Analyze the binary for the easy password*

Objective: *What is the password ?*

Tools: *Kali Linux on UTM (macOS M1), Ghidra*

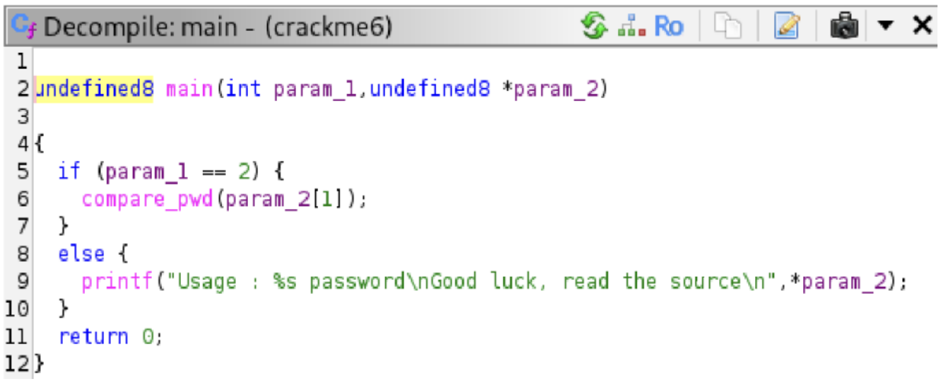
Let's not get intimitaded by previous file and see what awaits us after executing crackme6.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ ./crackme6
Usage : ./crackme6 password
Good luck, read the source
```

Picture 37: Executing crackme6.

As the author indicates, let's come back to Ghidra and import crackme6.

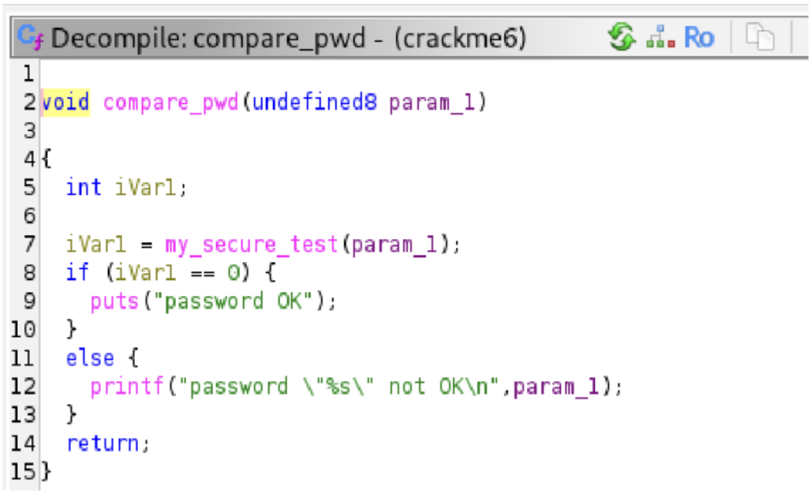


Decompile: main - (crackme6)

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

{
  if (param_1 == 2) {
    compare_pwd(param_2[1]);
  }
  else {
    printf("Usage : %s password\nGood luck, read the source\n",*param_2);
  }
  return 0;
}
```

Picture 38: main() function – Ghidra.

If we give one argument, then main() function passess the inputed argument (param_2[1]) to compare_pwd(). Let's see what's inside compare_pwd().

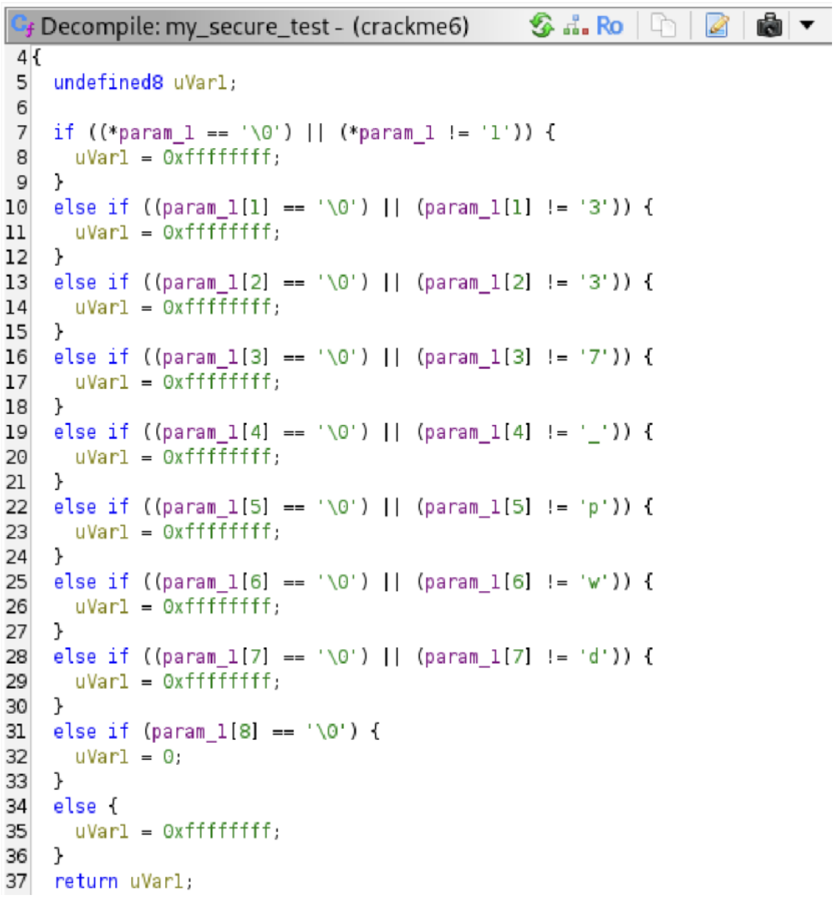


Decompile: compare_pwd - (crackme6)

```

void compare_pwd(undefined8 param_1)

{
  int iVar1;

  iVar1 = my_secure_test(param_1);
  if (iVar1 == 0) {
    puts("password OK");
  }
  else {
    printf("password \"%s\" not OK\n",param_1);
  }
  return;
}
```

Picture 39: compare_pwd() function – Ghidra.

We see that there's once again another function in use here: my_secure_test(). Let's go see inside.

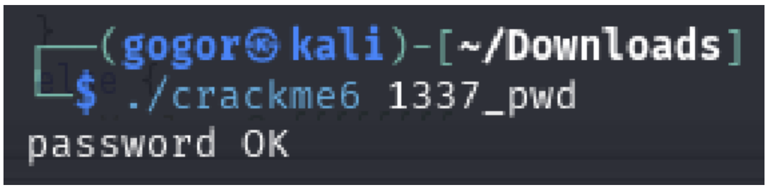


Decompile: my_secure_test - (crackme6)

```
{
  undefined8 uVar1;

  if ((*param_1 == '\0') || (*param_1 != '1')) {
    uVar1 = 0xffffffff;
  }
  else if ((param_1[1] == '\0') || (param_1[1] != '3')) {
    uVar1 = 0xffffffff;
  }
  else if ((param_1[2] == '\0') || (param_1[2] != '3')) {
    uVar1 = 0xffffffff;
  }
  else if ((param_1[3] == '\0') || (param_1[3] != '7')) {
    uVar1 = 0xffffffff;
  }
  else if ((param_1[4] == '\0') || (param_1[4] != '_')) {
    uVar1 = 0xffffffff;
  }
  else if ((param_1[5] == '\0') || (param_1[5] != 'p')) {
    uVar1 = 0xffffffff;
  }
  else if ((param_1[6] == '\0') || (param_1[6] != 'w')) {
    uVar1 = 0xffffffff;
  }
  else if ((param_1[7] == '\0') || (param_1[7] != 'd')) {
    uVar1 = 0xffffffff;
  }
  else if (param_1[8] == '\0') {
    uVar1 = 0;
  }
  else {
    uVar1 = 0xffffffff;
  }
  return uVar1;
```

Picture 40: my_secure_test() function – Ghidra.

Just by looking at those conditions, we see that by connecting those characters together we get ‘1337_pwd’.



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

Picture 41: Using the password to see if it’s correct.

Lucky us.

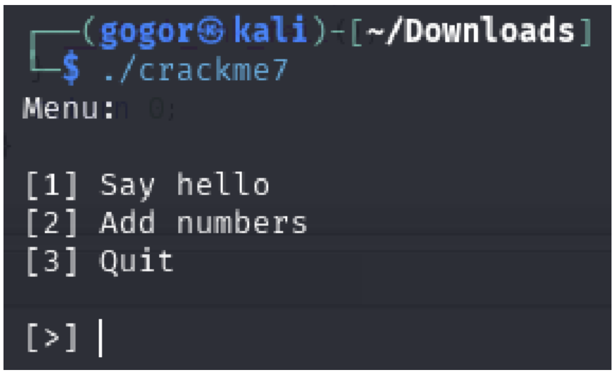
Crackme7

Description: *Analyze the binary to get the flag*

Objective: *What is the flag ?*

Tools: *Kali Linux on UTM (macOS M1), Ghidra, online decoder.*

Let’s get our hands on another file.

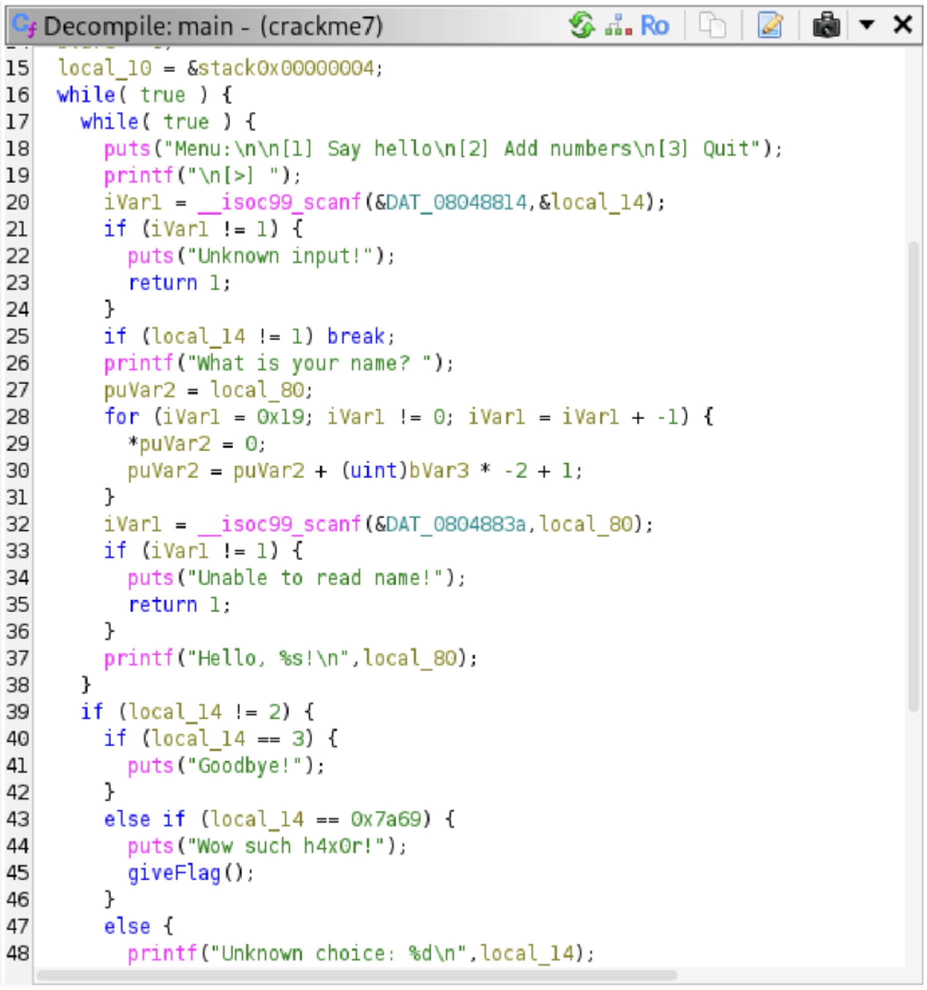


```
┌──(gogor㉿kali)-[~/Downloads]
└─$ ./crackme7
Menu:

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

[>] 
```

Picture 42: Executing crackme7.

Would you look at that! How fancy!

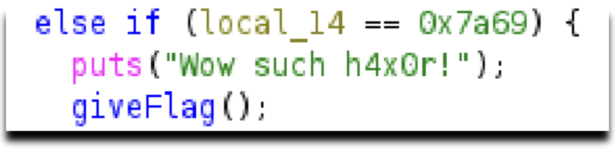
After trying different combinations, the program does what it says – print’s your name, adds numers or quits. My first suspection is that after entering certain number or string, we will get our flag. Let’s go!



Decompile: main - (crackme7)

```
  local_10 = &stack0x00000004;
  while( true ) {
    while( true ) {
      puts("Menu:\n\n[1] Say hello\n[2] Add numbers\n[3] Quit");
      printf("\n[>] ");
      iVar1 = __isoc99_scanf(&DAT_08048814,&local_14);
      if (iVar1 != 1) {
        puts("Unknown input!");
        return 1;
      }
      if (local_14 != 1) break;
      printf("What is your name? ");
      puVar2 = local_80;
      for (iVar1 = 0x19; iVar1 != 0; iVar1 = iVar1 + -1) {
        *puVar2 = 0;
        puVar2 = puVar2 + (uint)bVar3 * -2 + 1;
      }
      iVar1 = __isoc99_scanf(&DAT_0804883a,local_80);
      if (iVar1 != 1) {
        puts("Unable to read name!");
        return 1;
      }
      printf("Hello, %s!\n",local_80);
    }
    if (local_14 != 2) {
      if (local_14 == 3) {
        puts("Goodbye!");
      }
      else if (local_14 == 0x7a69) {
        puts("Wow such h4x0r!");
        giveFlag();
      }
      else {
        printf("Unknown choice: %d\n",local_14);
```

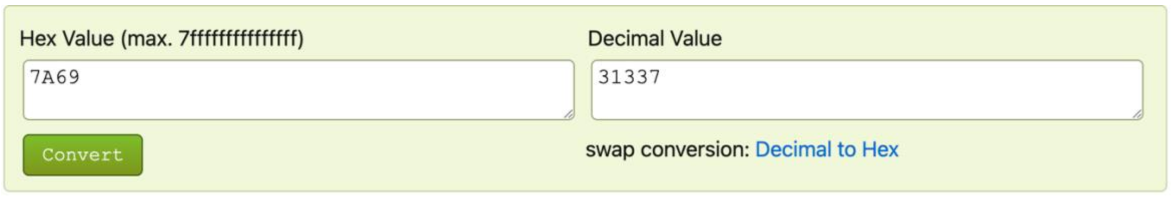
Picture 43: main() function – Ghidra.



```
else if (local_14 == 0x7a69) {
  puts("Wow such h4x0r!");
  giveFlag();
```

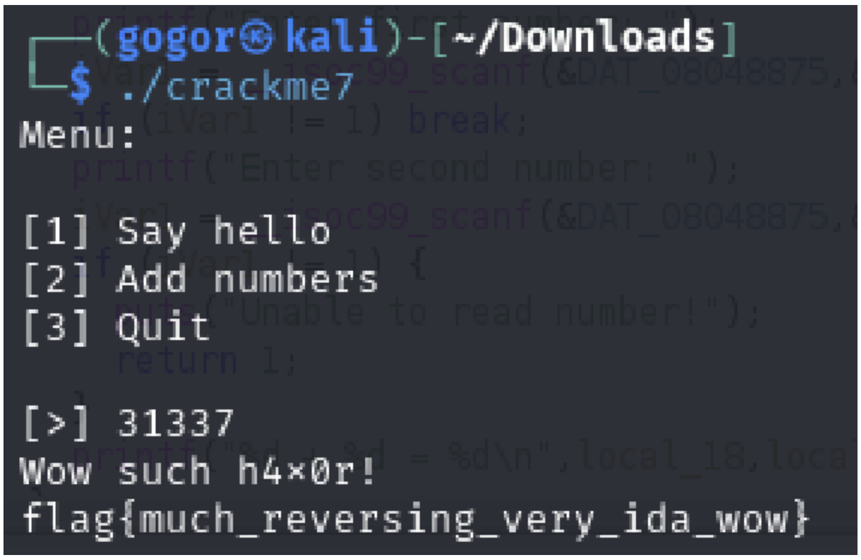
Picture 44: Interesting condition found in main() function – Ghidra.

After taking a quick look (you get really used to it after 4 previous tasks), local_l4 is our input, and if it's equal to this HEX value, we will get our flag. Let's see what's DEC value of this HEX using online decoder.



Hex Value (max. 7fffffffffffffff)	Decimal Value
7A69	31337

Picture 45: Converting HEX value to DEC.



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

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

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

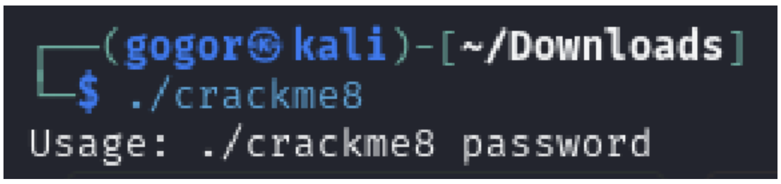
Picture 46: Entering the DEC into the menu of crackme7.

Crackme8

Description: Analyze the binary and obtain the flag

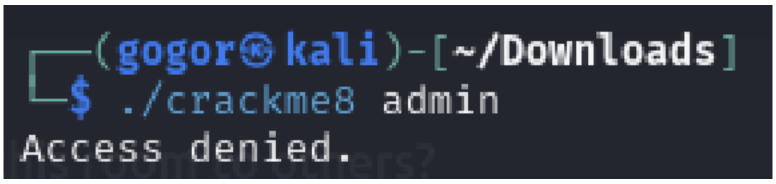
Objective: *What is the flag ?*

Tools: *Kali Linux on UTM (macOS M1), Ghidra*



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ ./crackme8
Usage: ./crackme8 password
```

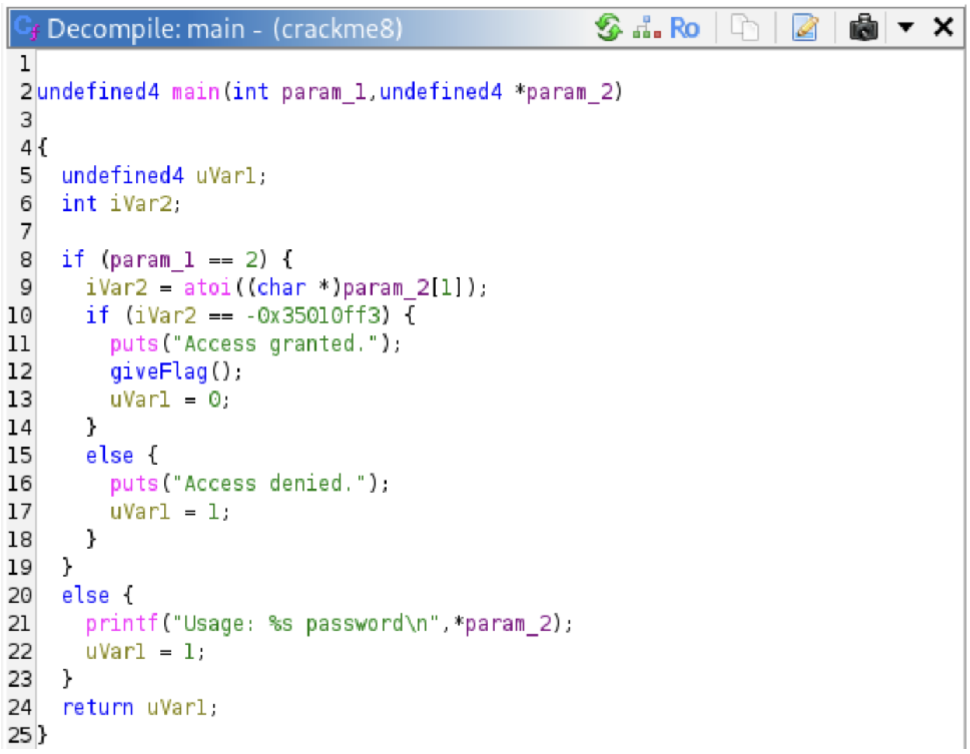
Picture 47: Executing crackme8.



```
┌──(gogor㉿kali)-[~/Downloads]
└─$ ./crackme8 admin
Access denied.
```

Picture 48: Mock-test of the program.

So simple that almost refreshing. Finding only a password? Let's get to it!

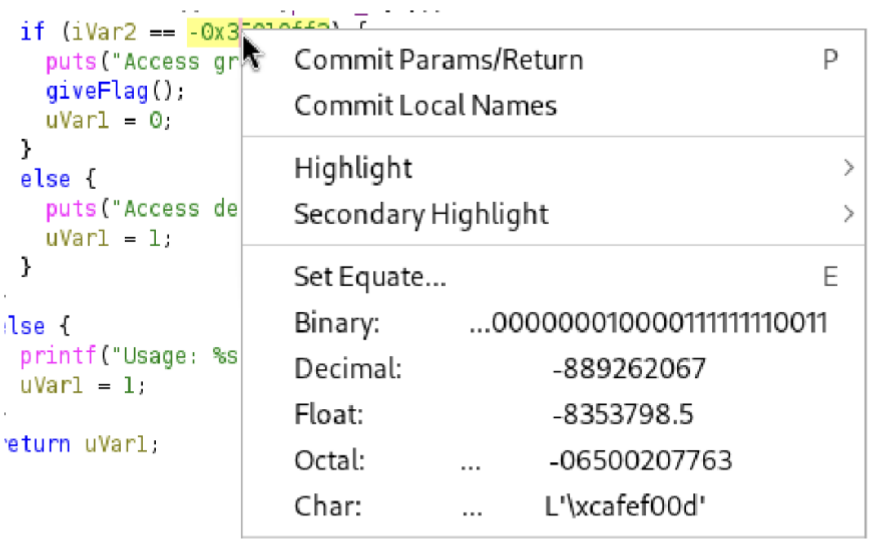


```
undefined4 main(int param_1,undefined4 *param_2)

{
  undefined4 uVar1;
  int iVar2;

  if (param_1 == 2) {
    iVar2 = atoi((char *)param_2[1]);
    if (iVar2 == -0x35010ff3) {
      puts("Access granted.");
      giveFlag();
      uVar1 = 0;
    }
    else {
      puts("Access denied.");
      uVar1 = 1;
    }
  }
  else {
    printf("Usage: %s password\n",*param_2);
    uVar1 = 1;
  }
  return uVar1;
}
```

Picture 49: main() function – Ghidra.

Okay, so we get access if our input is equal to this negative HEX value.

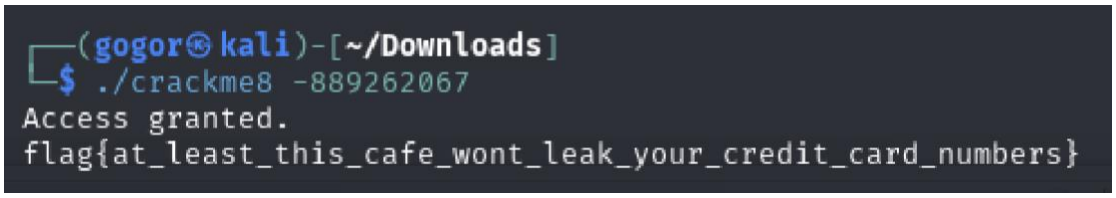
Let's take notice that our input is being messed with. It is converted into integer and being compared with that **negative** HEX value.



```
Binary:   ...00000001000011111111110011
Decimal:  -889262067
Float:    -8353798.5
Octal: ... -06500207763
Char:  ... L'\xcafef00d'
```

Picture 50: Ghidra's hover information.

Luckily, Ghidra already converts the value so we just need to retype *-889262067* as the argument.



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

Picture 51: Getting our flag.

This room sure was long and introduced many challanges for me, but they weren't disarming. I was able to quickly and eagerly search for answers and implement them with new tools. The best room I did so far.

I hope you had as much fun as me!

See you in another write-up!