TryHackMe Room Write-up

Title: 0x41haz

Description: Simple Reversing Challenge

Author: Igor Buszta

Find the password!

Description: *In this challenge, you are asked to solve a simple reversing solution. Download and analyze the binary to discover the password.*

There may be anti-reversing measures in place!

Objective: *What is the password?*

Tools*: Kali Linux, hexeditor, ghidra, hexdump, strings.*

Before we start with anything, let's gather more information about our downloaded file.

```
> file 0x41haz-1640335532346.0x41haz
0x41haz-1640335532346.0x41haz: ELF 64-bit MSB *unknown arch 0x3e00* (SYSV)
```

Picture 1: File information.

What can we tell right now?

ELF is Executable and Linkable Format – means Linux.

MSB is Most Significant Bit, indicates Big Endian encoding type.

There might be something wrong with this file - **unknown arch 0x3e00* (SYSV)*, it might be the anti-reversing measure mentioned in description. Let's now see what strings are hidden within the file.

```
> strings 0x41haz-1640335532346.0x41haz
/lib64/ld-linux-x86-64.so.2
gets
exit
puts
strlen
__cxa_finalize
__libc_start_main
libc.so.6
GLIBC_2.2.5
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
u/UH
2@@25$gfH
sT&@f
[]A\A]A^A_
=======================
Hey , Can You Crackme ?
=======================
It's jus a simple binary
Tell Me the Password :
Is it correct , I don't think so.
Nope
Well Done !!
;*3$"
```

Picture 2: Selected strings output.

If you know your architectures and file formats, you can notice that this file was build for x86-64 architecture, which uses *Little Endian encoding*.

Link on Endianess

> endianness is the dominant ordering for processor architectures (x86, most ARM implementations, base RISC-V implementations) and their associated memory. File formats can

Picture 3: Wiki page selected paragraph on which endianess is used on x86 architecture.

There's our problem! We get contradictory information: Big Endian used for x86 architecture? We must change it, but how? In order to do that, we should learn more about ELF header.

ELF header[5]

Offset		Size (bytes)		Field	Purpose
32-bit	64-bit	32-bit	64-bit		
0x00		4		e_ident[EI_MAG0] through e_ident[EI_MAG3]	0x7F followed by ELF (45 4c 46) in ASCII; these four bytes constitute the magic number.
0x04		1		e_ident[EI_CLASS]	This byte is set to either 1 or 2 to signify 32- or 64-bit format, respectively.
0x05		1		e_ident[EI_DATA]	This byte is set to either 1 or 2 to signify little or big endianness, respectively. This affects interpretation of multi-byte fields starting with offset 0x10.

Picture 3: Wiki page on ELF header contents.

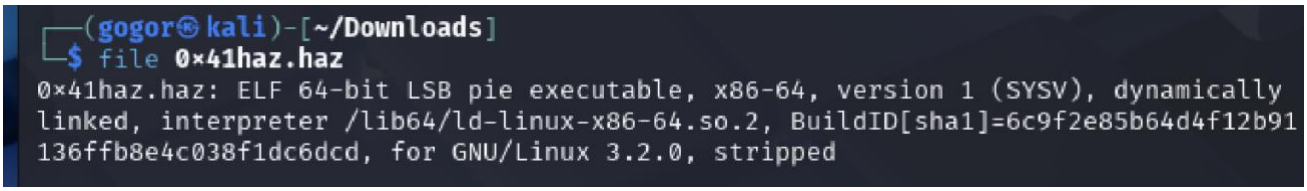
First four bytes are used to determine the *magic number*, fifth byte is used to signify 32- or 64-bit format. Sixth byte is our goal. It should be changed from big endian (2) to little endian (1).

Let's use hexeditor to make changes.

Picture 4: ELF header before change.

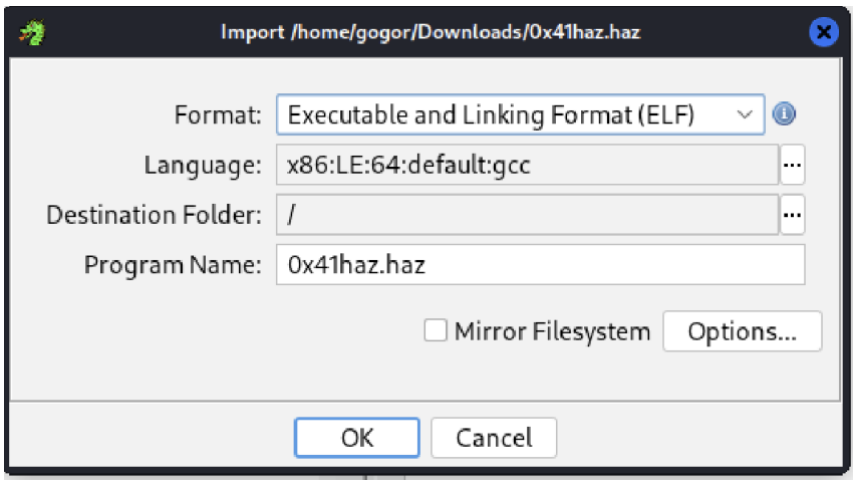
Picture 5: ELF header after change.

Now we can see if everything is correct now.



Picture 6: file information after changing the endian byte.

Now we can reverse engineer this bad boy. I used Ghidra.



Picture 7: Importing the file to Ghidra.

I went through the functions and in function FUN_00101165 I found something juicy,

Decompile: FUN_00101165 - (0x41haz.haz)

```
1
2  undefined8 FUN_00101165(void)
3
4  {
5    size_t sVar1;
6    char local_48 [42];
7    char local_1e [14];
8    int local_10;
9    int local_c;
10
11   __builtin_strncpy(local_1e,"2@@25$gfsT&@L",0xe);
12   puts("======================\nHey , Can You Crackme ?\n======================");
13   puts("It\'s jus a simple binary \n");
14   puts("Tell Me the Password :");
15   gets(local_48);
16   sVar1 = strlen(local_48);
17   local_10 = (int)sVar1;
18   if ((int)sVar1 != 0xd) {
19     puts("Is it correct , I don\'t think so.");
20                     /* WARNING: Subroutine does not return */
21     exit(0);
22   }
23   local_c = 0;
24   while( true ) {
25     if (0xc < local_c) {
26       puts("Well Done !!");
27       return 0;
28     }
29     if (local_1e[local_c] != local_48[local_c]) break;
30     local_c = local_c + 1;
31   }
32   puts("Nope");
33                     /* WARNING: Subroutine does not return */
34   exit(0);
35 }
36
```

Picture 7: FUN_00101165 function contents.

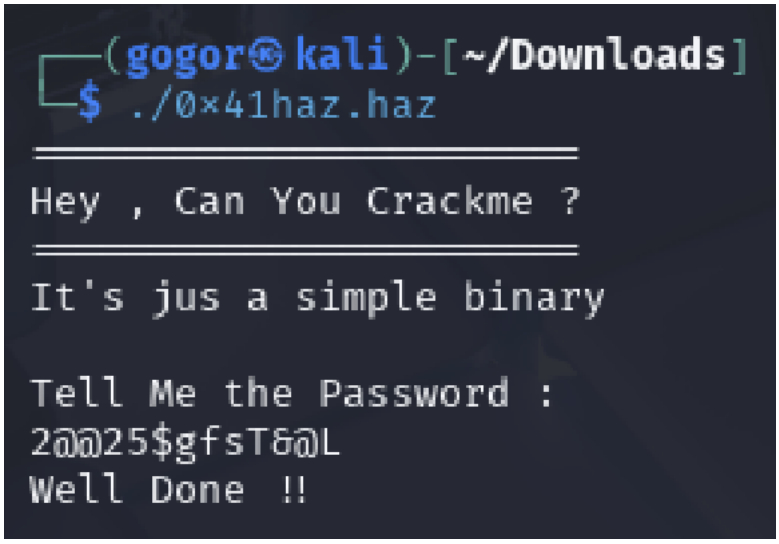
Let's focus on where the program gets our input, where it compares it and to what string.

```
__builtin_strncpy(local_1e,"2@@25$gfsT&@L",0xe);
```

Picture 8: Fragment of interesting code.

Why is line on Picture 8 so interesting? Because it has something what looks like flag. After we try it, it is indeed what we're looking for, but why?

Program checks the lenght of our input (while (true) loop) and the contents (if condition on line 29) character by character. If any of the sign in our input is different than the one in local_le (variable to which the flag is copied to), the function breaks.



Picture 9: Executing the code with wanted flag.