CTF Writeup: 0x41haz (Tryhackme)

BY: CHANDNI CHANDRASHEKAR
DATED: 12-07-2025

1. 🎯 **Challenge Overview:**

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

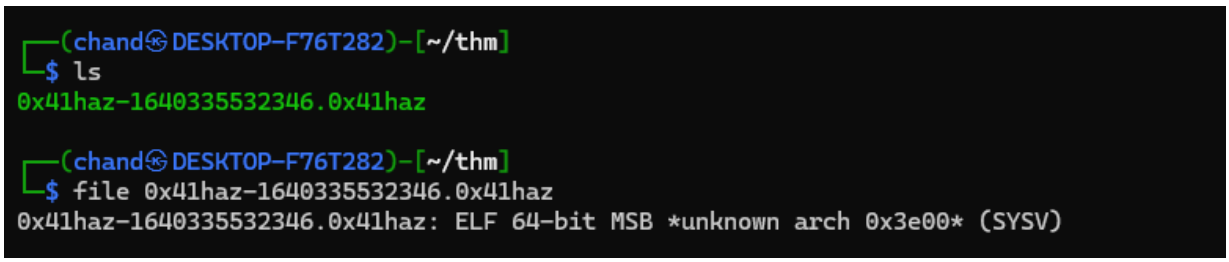
2. 🛠️ **Tools Used:**

TOOLS/COMMAND	PURPOSE
file	To identify binary type (eg. ELF)
strings	To extract readable ASCII strings from bin
objdump -d	To dissemble binary to assembly
Cyberchef	Decode little endian to ASCII
Hexed.it	Online hex editor
Kali Linux (WSL)	Terminal – command line
Tryhackme	Host room

3. 🔍 **Step-by-Step Breakdown:**

STEP 1: Analysing the binary (file downloaded from THM) using file command.

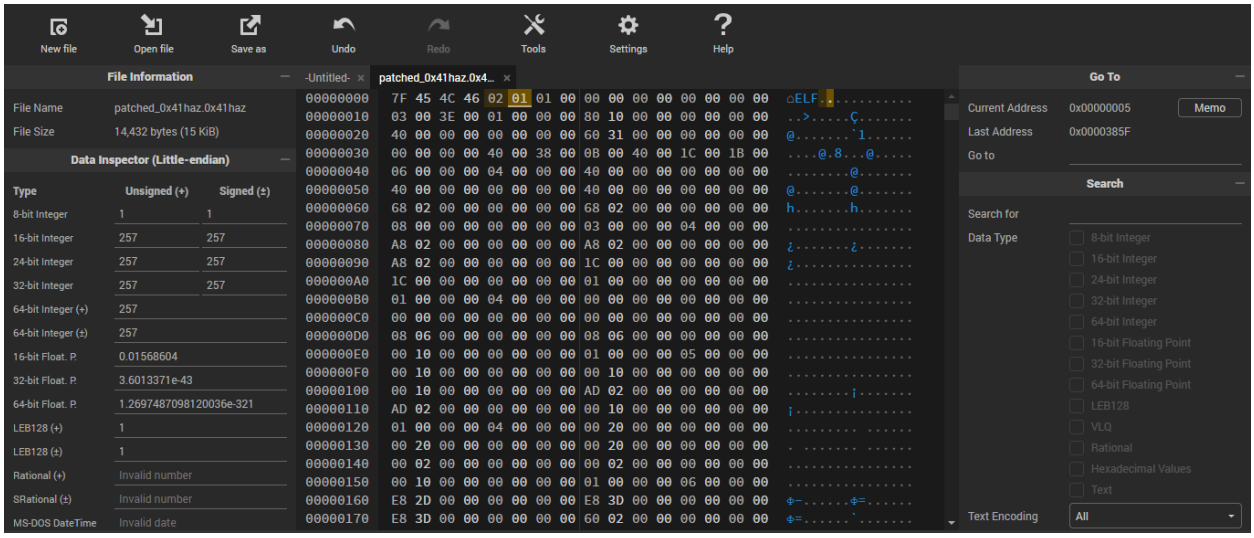
- Gave an ELF unknown arch executable file as its output. (screenshot below)



- **What I learned:**
  The binary is compiled for 64-bit architecture but in **MSB (Big Endian)** format, which Kali can't read properly — it needs **LSB (Little Endian)**.

STEP 2: Patched Endianness with hexed.it.

- Opened the .0x41haz file on hexed.it and at byte offset (0x05) the value 02 (MSB) was changed to 01 (LSB) so that the file is readable by kali and can be interpreted by certain tools. (screenshot below)



- Verifying the patch and it gave the result as an ELF executable file. Hence patch was successful, allowing me to use tools. (screenshot below)

```
┌──(chand㉿DESKTOP-F76T282)-[~/thm]
└─$ ls
0x41haz-1640335532346.0x41haz  patched_0x41haz

┌──(chand㉿DESKTOP-F76T282)-[~/thm]
└─$ file patched_0x41haz
patched_0x41haz: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]
=6c9f2e85b64d4f12b91136ffb8e4c038f1dc6dcd, for GNU/Linux 3.2.0, stripped
```

STEP 3: Exploring Strings:

- Ran the **strings** command on the patched file – to lookout for more hints/hidden ASCII strings within the binary file – came across this: (screenshot below)

```
chand@DESKTOP-F76T282: ~

┌──(chand㉿DESKTOP-F76T282)-[~/thm]
└─$ file patched_0x41haz
patched_0x41haz: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]
=6c9f2e85b64d4f12b91136ffb8e4c038f1dc6dcd, for GNU/Linux 3.2.0, stripped

┌──(chand㉿DESKTOP-F76T282)-[~/thm]
└─$ strings patched_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$gfH
sT&@f
[]A\A]A^A_
========================
Hey , Can You Crackme ?
========================
It's jus a simple binary
Tell Me the Password :
Is it correct , I don't think so.
Nope
Well Done !!
;*3$"
GCC: (Debian 10.3.0-9) 10.3.0
.shstrtab
```

- This confirmed that the binary performs a password check — suggesting that the correct password must be stored or validated internally.

STEP 4: Disassembling with objdump command:

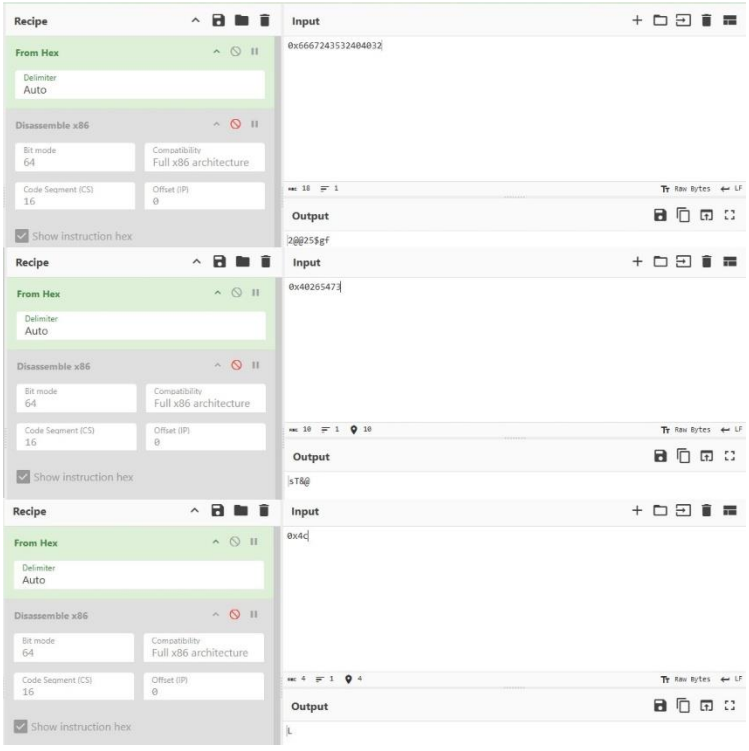
- Used **objdump** to disassemble the binary and reveal its low-level assembly instructions, allowing us to find hardcoded values (like the password) directly in the code. (screenshot below)

```
chand@DESKTOP-F76T282: ~
┌──(chand㉿DESKTOP-F76T282)-[~/thm]
└─$ objdump -d patched_0x41haz

patched_0x41haz:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <.init>:
    1000:       48 83 ec 08             sub    $0x8,%rsp
    1004:       48 8b 05 dd 2f 00 00    mov    0x2fdd(%rip),%rax        # 3fe8 <__cxa_finalize@plt+0x2f78>
    100b:       48 85 c0                test   %rax,%rax
    100e:       74 02                   je     1012 <puts@plt-0x1e>
    1010:       ff d0                   call   *%rax
    1012:       48 83 c4 08             add    $0x8,%rsp
    1016:       c3                      ret

Disassembly of section .plt:

0000000000001020 <puts@plt-0x10>:
    1020:       ff 35 e2 2f 00 00       push   0x2fe2(%rip)        # 4008 <__cxa_finalize@plt+0x2f98>
    1026:       ff 25 e4 2f 00 00       jmp    *0x2fe4(%rip)        # 4010 <__cxa_finalize@plt+0x2fa0>
    102c:       0f 1f 40 00             nopl   0x0(%rax)

0000000000001030 <puts@plt>:
    1030:       ff 25 e2 2f 00 00       jmp    *0x2fe2(%rip)        # 4018 <__cxa_finalize@plt+0x2fa8>
    1036:       68 00 00 00 00          push   $0x0
    103b:       e9 e0 ff ff ff          jmp    1020 <puts@plt-0x10>

0000000000001040 <strlen@plt>:
    1040:       ff 25 da 2f 00 00       jmp    *0x2fda(%rip)        # 4020 <__cxa_finalize@plt+0x2fb0>
    1046:       68 01 00 00 00          push   $0x1
    104b:       e9 d0 ff ff ff          jmp    1020 <puts@plt-0x10>

0000000000001050 <gets@plt>:
```

- ***Key findings:*** movabs $0x6667243532404032, %rax, movl $0x40265473, -0xe(%rbp), and movw $0x4c,-0xa(%rbp) – likely a password.

STEP 5: Decoding the password with Cyberchef:

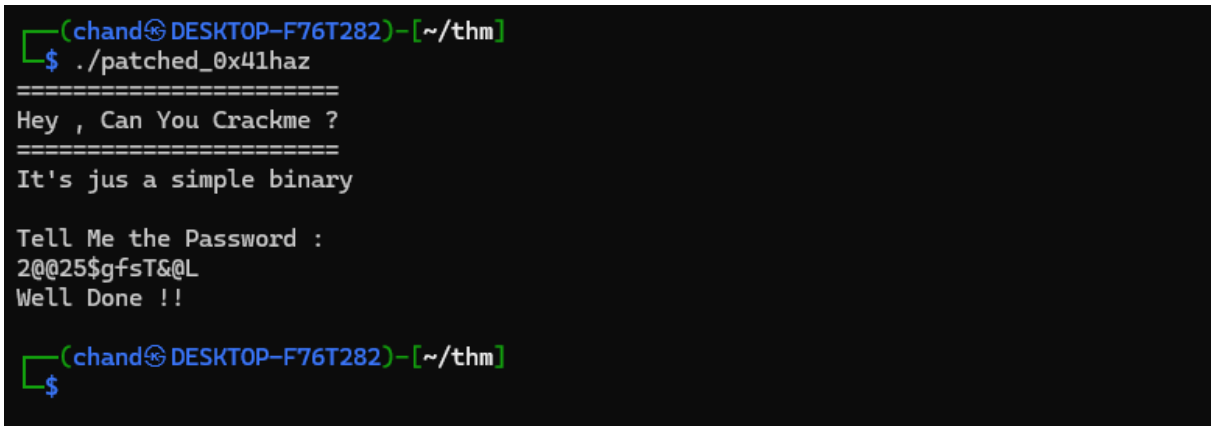
- Reversed hex in little endian order.
- Converted to standard key using Cyberchef. (screenshot below)



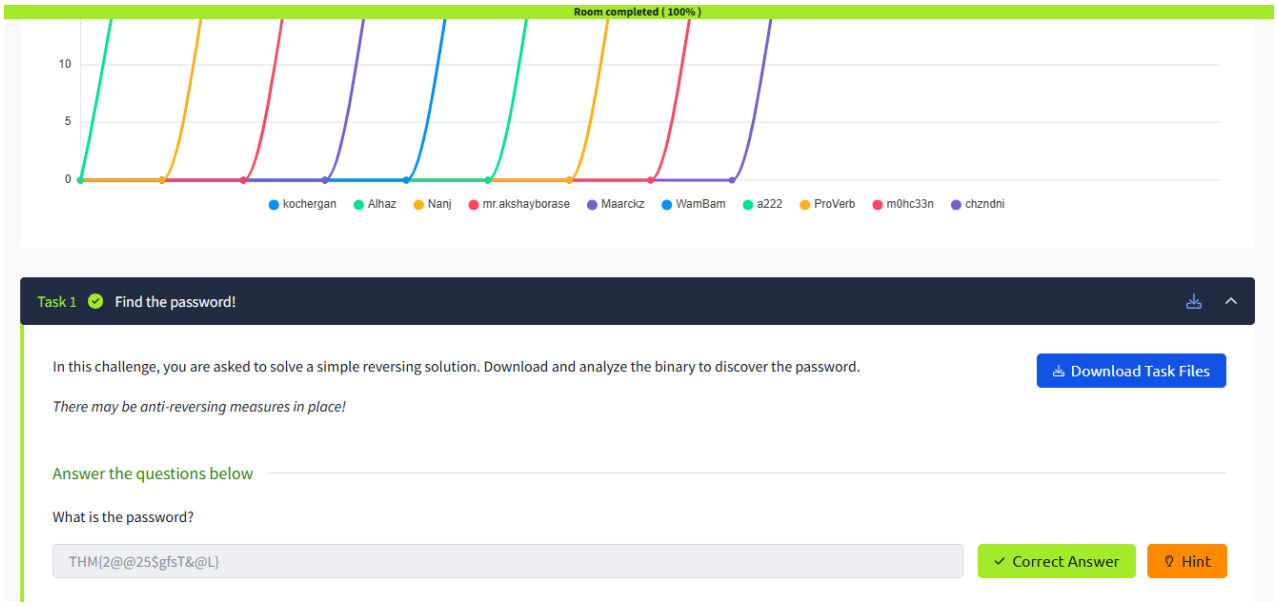
Final Password: 2@@25$gfsT&@L

STEP 7: Testing the password:

- Successful output has been obtained. (screenshot below)



On THM: Room has been completed successfully. (screenshot below)



What I Learned

- How to patch and interpret ELF binaries.
- Difference between Big Endian and Little Endian.
- Using “objdump” to reverse engineer binary logic.
- How values are stored and compared in memory.
- Confidence in using CyberChef and hex editors.