Battle of Hacker (BOH) 2021

Reverse Engineering Writeups

Written by: Muhammad Zhafran || FPanda

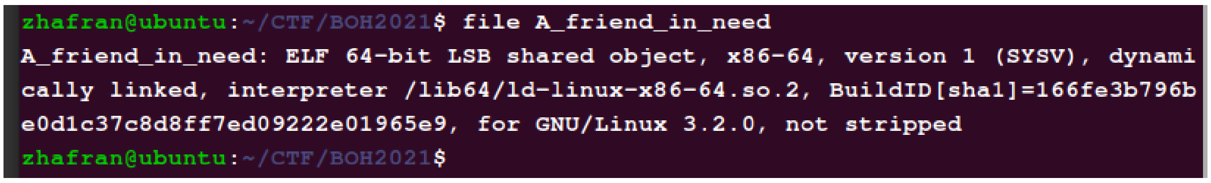
Table of Contents

1 – A friend in need

Description:

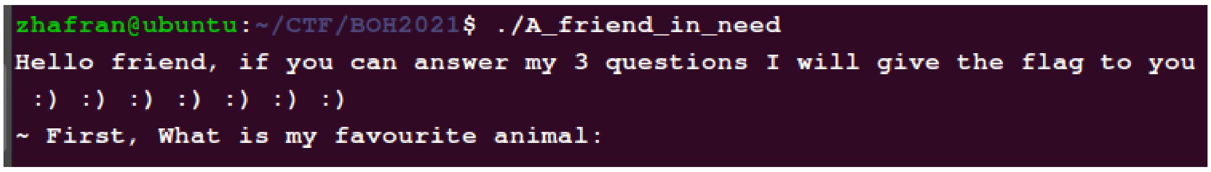
“Lucky you!!! You have such a good friend that will give you the flag for free.... or is it?”

Using File command showed that it is a ELF-64 Bit file.



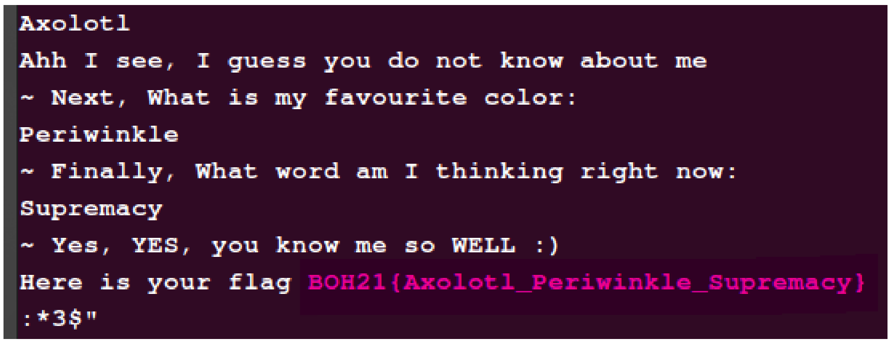
```
zhafran@ubuntu:~/CTF/BOH2021$ file A_friend_in_need
A_friend_in_need: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynami
cally linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=166fe3b796b
e0d1c37c8d8ff7ed09222e01965e9, for GNU/Linux 3.2.0, not stripped
zhafran@ubuntu:~/CTF/BOH2021$
```

Running the binary will prompt for an input:



```
zhafran@ubuntu:~/CTF/BOH2021$ ./A_friend_in_need
Hello friend, if you can answer my 3 questions I will give the flag to you
 :) :) :) :) :) :) :)
~ First, What is my favourite animal:
```

We can use **Radare2** or other tools to discover for the right input, or we can try printing all of the strings **using Linux strings command**.



```
Axolotl
Ahh I see, I guess you do not know about me
~ Next, What is my favourite color:
Periwinkle
~ Finally, What word am I thinking right now:
Supremacy
~ Yes, YES, you know me so WELL :)
Here is your flag BOH21{Axolotl_Periwinkle_Supremacy}
:*3$"
```

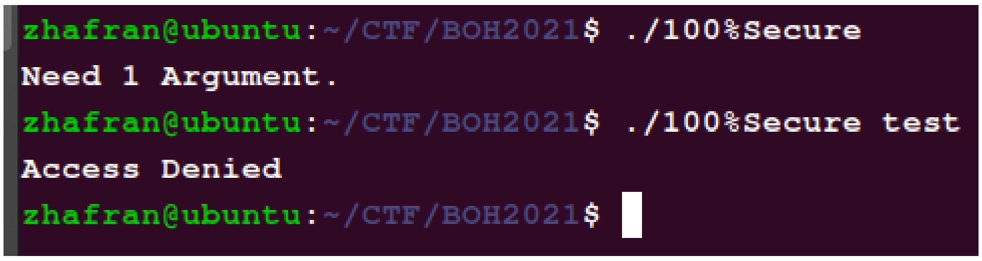
Flag: BOH21{Axolotl_Periwinkle_Supremacy}

2 – 100%Secure

Description:

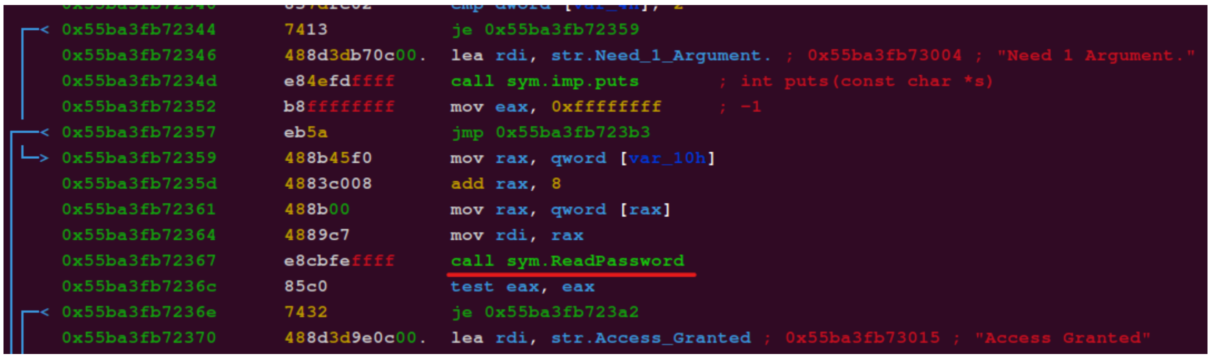
"I hide the flag inside this highly secure binary that no one in this world can ever break it."

Running the binary will ask for an argument, wrong argument will give **Access Denied** output.



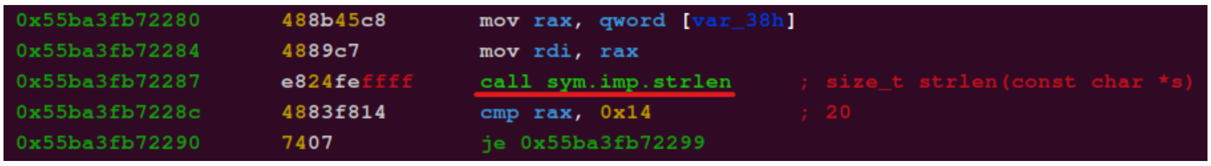
```
zhafran@ubuntu:~/CTF/BOH2021$ ./100%Secure
Need 1 Argument.
zhafran@ubuntu:~/CTF/BOH2021$ ./100%Secure test
Access Denied
zhafran@ubuntu:~/CTF/BOH2021$
```

By using Radare2, inside the program **main** function shows that the function will call another function called **ReadPassword**. It will pass the user input argument as another argument to the ReadPassword function.



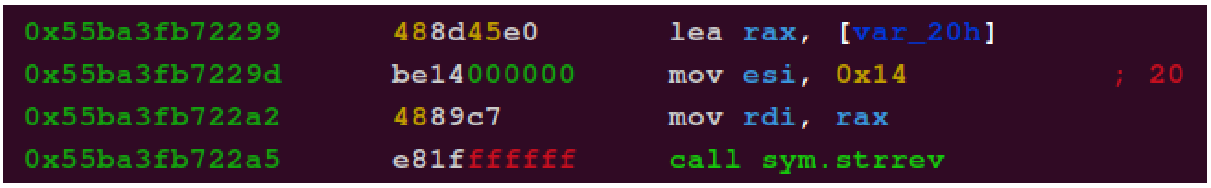
```
┌─< 0x55ba3fb72344      7413           je 0x55ba3fb72359
│   0x55ba3fb72346      488d3db70c00.  lea rdi, str.Need_1_Argument. ; 0x55ba3fb73004 ; "Need 1 Argument."
│   0x55ba3fb7234d      e84efdffff     call sym.imp.puts          ; int puts(const char *s)
│   0x55ba3fb72352      b8ffffffff     mov eax, 0xffffffff        ; -1
┌──< 0x55ba3fb72357     eb5a           jmp 0x55ba3fb723b3
│└─> 0x55ba3fb72359     488b45f0       mov rax, qword [var_10h]
│   0x55ba3fb7235d      4883c008       add rax, 8
│   0x55ba3fb72361      488b00         mov rax, qword [rax]
│   0x55ba3fb72364      4889c7         mov rdi, rax
│   0x55ba3fb72367      e8cbfeffff     call sym.ReadPassword
│   0x55ba3fb7236c      85c0           test eax, eax
│┌─< 0x55ba3fb7236e     7432           je 0x55ba3fb723a2
││  0x55ba3fb72370      488d3d9e0c00.  lea rdi, str.Access_Granted ; 0x55ba3fb73015 ; "Access Granted"
```

Inside the **ReadPassword** function we can see that it calls **strlen** function which will check the length of the user input argument. Then it will compare the result with 20. If it is equal it will continue, if not it will print "Access Denied". **At this point we know that the correct input length is 20 characters.**



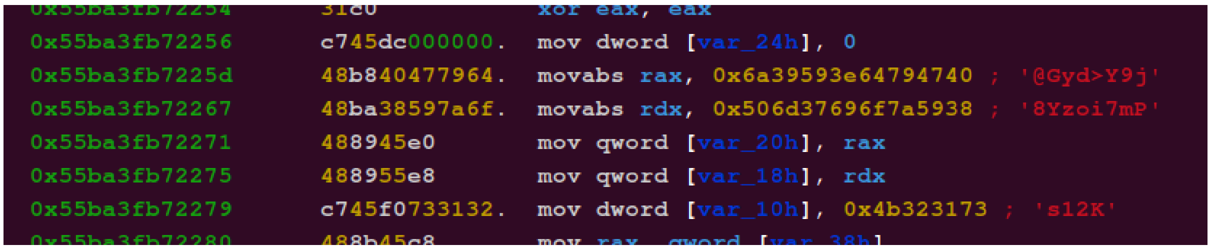
```
0x55ba3fb72280      488b45c8       mov rax, qword [var_38h]
0x55ba3fb72284      4889c7         mov rdi, rax
0x55ba3fb72287      e824feffff     call sym.imp.strlen      ; size_t strlen(const char *s)
0x55ba3fb7228c      4883f814       cmp rax, 0x14            ; 20
0x55ba3fb72290      7407           je 0x55ba3fb72299
```

Then it will call **strrev function** which use offset variable [var_20h] (rbp-0x20) as their argument. The function itself is to reverse the strings of the passed argument.



```
0x55ba3fb72299      488d45e0       lea rax, [var_20h]
0x55ba3fb7229d      be14000000     mov esi, 0x14          ; 20
0x55ba3fb722a2      4889c7         mov rdi, rax
0x55ba3fb722a5      e81fffffff     call sym.strrev
```

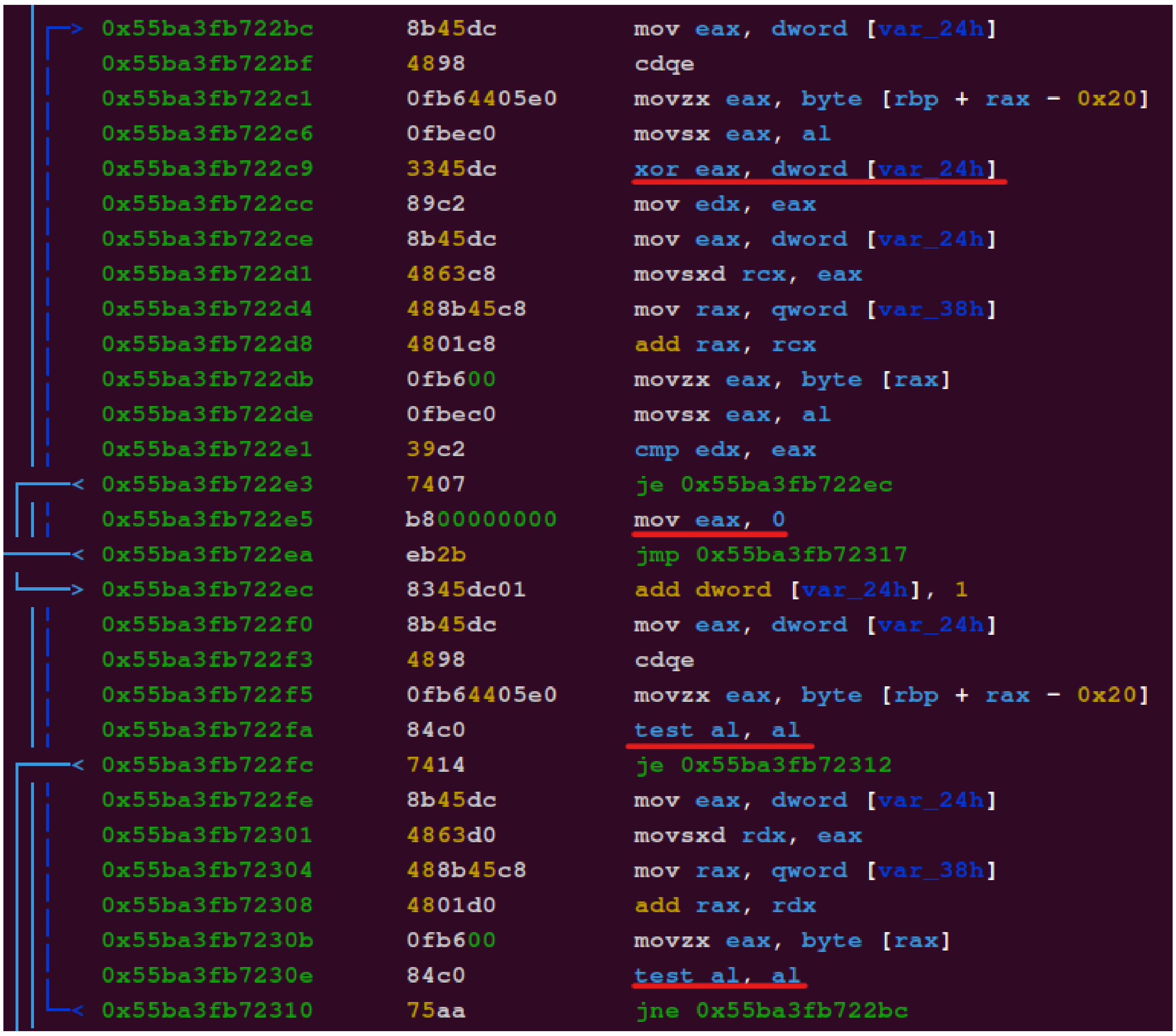
Therefore, before the function call, var_20h is "@Gyd>Y9j8Yzoi7mPs12K" after strrev it will be "K21sPm7iozY8j9Y>dyG@".



```
0x55ba3fb72254      31c0          xor eax, eax
0x55ba3fb72256      c745dc000000.  mov dword [var_24h], 0
0x55ba3fb7225d      48b840477964.  movabs rax, 0x6a39593e64794740 ; '@Gyd>Y9j'
0x55ba3fb72267      48ba38597a6f.  movabs rdx, 0x506d37696f7a5938 ; '8Yzoi7mP'
0x55ba3fb72271      488945e0      mov qword [var_20h], rax
0x55ba3fb72275      488955e8      mov qword [var_18h], rdx
0x55ba3fb72279      c745f0733132.  mov dword [var_10h], 0x4b323173 ; 's12K'
0x55ba3fb72280      488b45c8      mov rax, qword [var_38h]
```

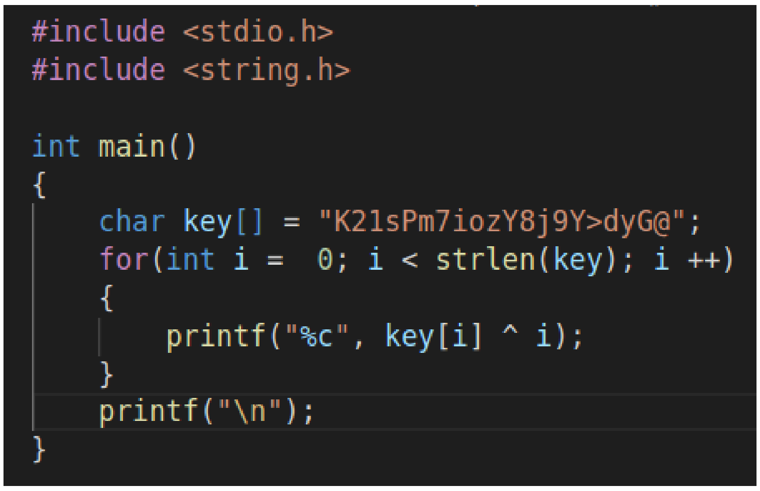
Back to ReadPassword function, the program continues and judging by the amount of jump instruction and the existence of loop counter which is var_24h that is continuously being incremented by one, we can say that this is a **loop function**.

The program will then take the **reversed var_20h index by index and will do a XOR with var_24h (loop counter, initially 0)**. Then it will **compare the result with the user input argument (var_38h)**. If result is wrong or different it will move 0 to eax which means it will return 0, otherwise it will continue on looping and at the end it will move 1 to eax which means the program will return 1.



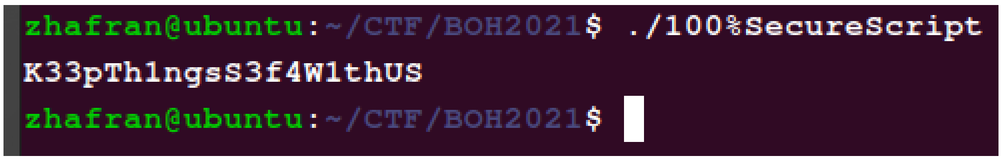
```
    ┌─> 0x55ba3fb722bc      8b45dc          mov eax, dword [var_24h]
    ╎   0x55ba3fb722bf      4898            cdqe
    ╎   0x55ba3fb722c1      0fb64405e0      movzx eax, byte [rbp + rax - 0x20]
    ╎   0x55ba3fb722c6      0fbec0          movsx eax, al
    ╎   0x55ba3fb722c9      3345dc          xor eax, dword [var_24h]
    ╎   0x55ba3fb722cc      89c2            mov edx, eax
    ╎   0x55ba3fb722ce      8b45dc          mov eax, dword [var_24h]
    ╎   0x55ba3fb722d1      4863c8          movsxd rcx, eax
    ╎   0x55ba3fb722d4      488b45c8        mov rax, qword [var_38h]
    ╎   0x55ba3fb722d8      4801c8          add rax, rcx
    ╎   0x55ba3fb722db      0fb600          movzx eax, byte [rax]
    ╎   0x55ba3fb722de      0fbec0          movsx eax, al
    ╎   0x55ba3fb722e1      39c2            cmp edx, eax
  ┌───< 0x55ba3fb722e3      7407            je 0x55ba3fb722ec
  │ ╎   0x55ba3fb722e5      b800000000      mov eax, 0
 ─────< 0x55ba3fb722ea      eb2b            jmp 0x55ba3fb72317
  └───> 0x55ba3fb722ec      8345dc01        add dword [var_24h], 1
    ╎   0x55ba3fb722f0      8b45dc          mov eax, dword [var_24h]
    ╎   0x55ba3fb722f3      4898            cdqe
    ╎   0x55ba3fb722f5      0fb64405e0      movzx eax, byte [rbp + rax - 0x20]
    ╎   0x55ba3fb722fa      84c0            test al, al
  ┌───< 0x55ba3fb722fc      7414            je 0x55ba3fb72312
  │ ╎   0x55ba3fb722fe      8b45dc          mov eax, dword [var_24h]
  │ ╎   0x55ba3fb72301      4863d0          movsxd rdx, eax
  │ ╎   0x55ba3fb72304      488b45c8        mov rax, qword [var_38h]
  │ ╎   0x55ba3fb72308      4801d0          add rax, rdx
  │ ╎   0x55ba3fb7230b      0fb600          movzx eax, byte [rax]
  │ ╎   0x55ba3fb7230e      84c0            test al, al
  │ └─< 0x55ba3fb72310      75aa            jne 0x55ba3fb722bc
```

We can either xor the reversed key one by one or create a script to do it.



```c
#include <stdio.h>
#include <string.h>

int main()
{
    char key[] = "K21sPm7iozY8j9Y>dyG@";
    for(int i =  0; i < strlen(key); i ++)
    {
        printf("%c", key[i] ^ i);
    }
    printf("\n");
}
```

Simple C script



```
zhafran@ubuntu:~/CTF/BOH2021$ ./100%SecureScript
K33pTh1ngsS3f4W1thUS
zhafran@ubuntu:~/CTF/BOH2021$ 
```

The output

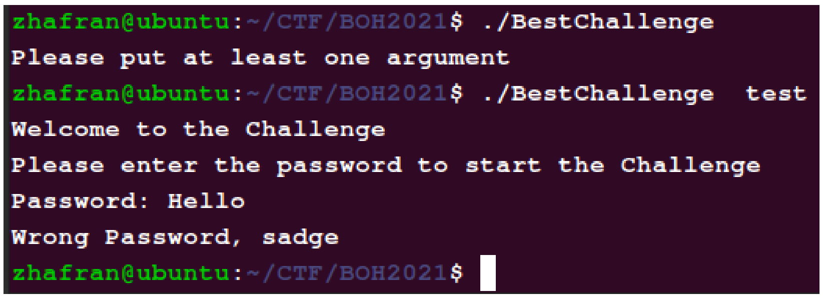
Flag: BOH21{K33pTh1ngsS3f4W1thUS}

3 – BestChallenge

Description:

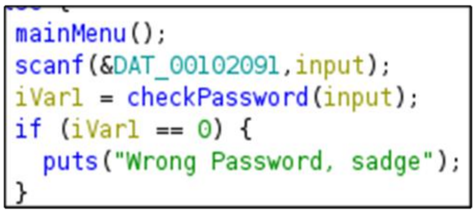
"Welcome to BOH21, I created this binary just for you, and maybe I hide a things or two inside it :)"

Running the challenge will prompt the User for an argument. After running it again with an argument this time it will ask for a password. Apparently if the password is false it will give "**Wrong Password, sadge"** output.



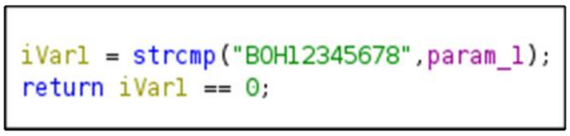
```
zhafran@ubuntu:~/CTF/BOH2021$ ./BestChallenge
Please put at least one argument
zhafran@ubuntu:~/CTF/BOH2021$ ./BestChallenge  test
Welcome to the Challenge
Please enter the password to start the Challenge
Password: Hello
Wrong Password, sadge
zhafran@ubuntu:~/CTF/BOH2021$ 
```

Using Ghidra we can decompile the binary into a rough pseudo C code.



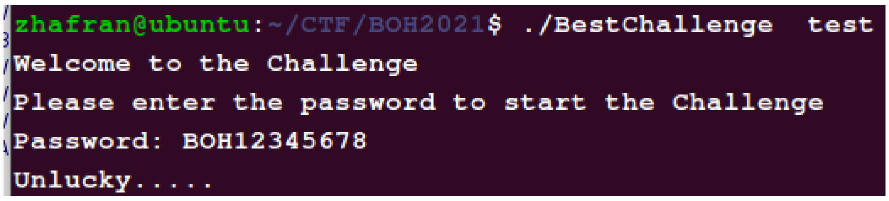
```
mainMenu();
scanf(&DAT_00102091,input);
iVar1 = checkPassword(input);
if (iVar1 == 0) {
  puts("Wrong Password, sadge");
}
```

Inside the main function we can see it use scanf to get user input and later on will use the variable from scanf as an argument to checkPassword function. Inside the checkPassword we can see that it compares with strings "BOH12345678". **So now we know the password is BOH12345678.**



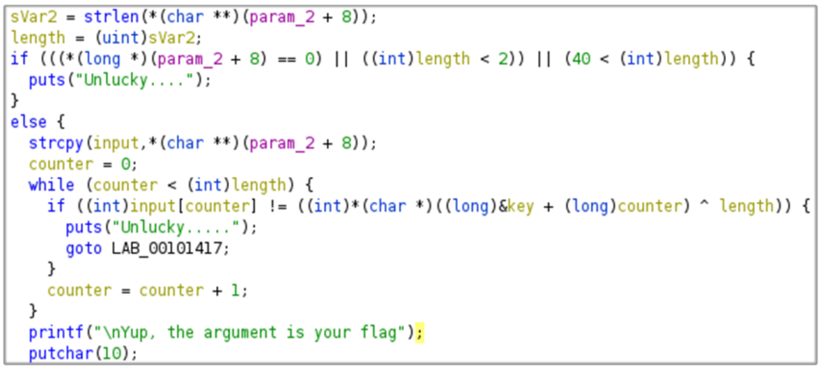
```
iVar1 = strcmp("BOH12345678",param_1);
return iVar1 == 0;
```

However, after we put in "BOH12345678" as the password it still does not give us the flag.



```
zhafran@ubuntu:~/CTF/BOH2021$ ./BestChallenge  test
Welcome to the Challenge
Please enter the password to start the Challenge
Password: BOH12345678
Unlucky.....
```

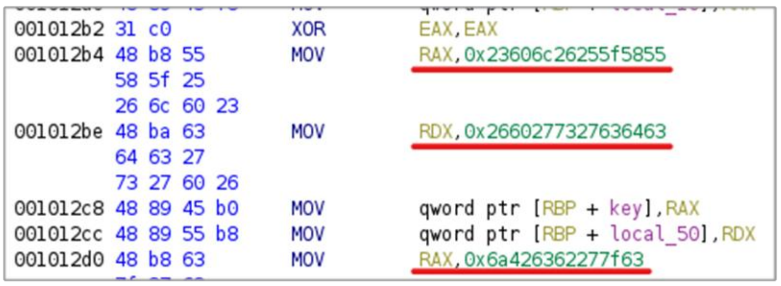
Further reversing shows that the program gets the length of our argument using **strlen**. It will then check if the result is between 2 to 40. If it match it will move the strings from our arguments into a different variable (we name this **variable input**) using strcpy. It will then compare our input with variable key XORed with the length of our input. If it does not match it will quit else it will continue on looping.



```
sVar2 = strlen(*(char **)(param_2 + 8));
length = (uint)sVar2;
if (((*(long *)(param_2 + 8) == 0) || ((int)length < 2)) || (40 < (int)length)) {
  puts("Unlucky....");
}
else {
  strcpy(input,*(char **)(param_2 + 8));
  counter = 0;
  while (counter < (int)length) {
    if ((int)input[counter] != ((int)*(char *)((long)&key + (long)counter) ^ length)) {
      puts("Unlucky.....");
      goto LAB_00101417;
    }
    counter = counter + 1;
  }
  printf("\nYup, the argument is your flag");
  putchar(10);
```

Since the key is most probably the obfuscated flag, what we can do is to take the key and XOR it with their own length.

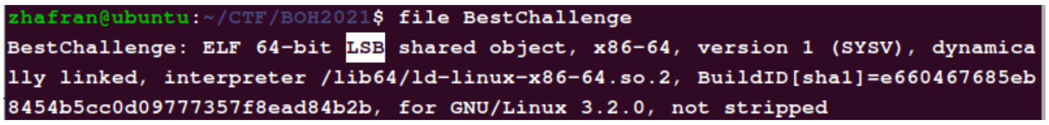
The value of the key variable can be found at the beginning of the main function.



```
001012b2 31 c0          XOR      EAX,EAX
001012b4 48 b8 55       MOV      RAX,0x23606c26255f5855
         58 5f 25
         26 6c 60 23
001012be 48 ba 63       MOV      RDX,0x2660277327636463
         64 63 27
         73 27 60 26
001012c8 48 89 45 b0    MOV      qword ptr [RBP + key],RAX
001012cc 48 89 55 b8    MOV      qword ptr [RBP + local_50],RDX
001012d0 48 b8 63       MOV      RAX,0x6a426362277f63
```

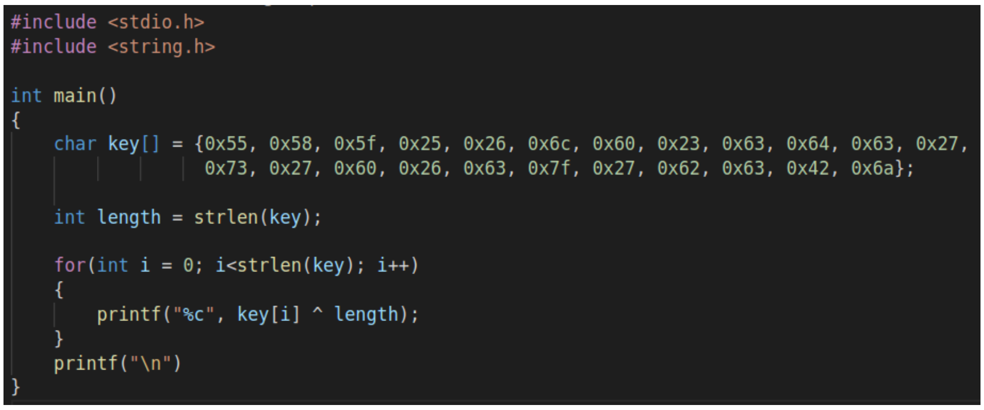
And since the binary is in little endian format it means that the Least Significant Bit (LSB) is stored upfront.

Therefore, the value of key should be **"0x55, 0x58, 0x5f, 0x25, 0x26, 0x6c, 0x60, 0x23, 0x63, 0x64, 0x63, 0x27, 0x73, 0x27, 0x60, 0x26, 0x63, 0x7f, 0x27, 0x62, 0x63, 0x42, 0x6a"**

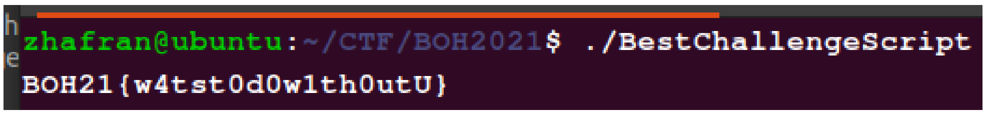


```
zhafran@ubuntu:~/CTF/BOH2021$ file BestChallenge
BestChallenge: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamica
lly linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=e660467685eb
8454b5cc0d09777357f8ead84b2b, for GNU/Linux 3.2.0, not stripped
```

A simple C script should be able to do the job to get the flag.



```c
#include <stdio.h>
#include <string.h>

int main()
{
    char key[] = {0x55, 0x58, 0x5f, 0x25, 0x26, 0x6c, 0x60, 0x23, 0x63, 0x64, 0x63, 0x27,
                0x73, 0x27, 0x60, 0x26, 0x63, 0x7f, 0x27, 0x62, 0x63, 0x42, 0x6a};

    int length = strlen(key);

    for(int i = 0; i<strlen(key); i++)
    {
        printf("%c", key[i] ^ length);
    }
    printf("\n")
}
```



```
zhafran@ubuntu:~/CTF/BOH2021$ ./BestChallengeScript
BOH21{w4tst0d0w1th0utU}
```

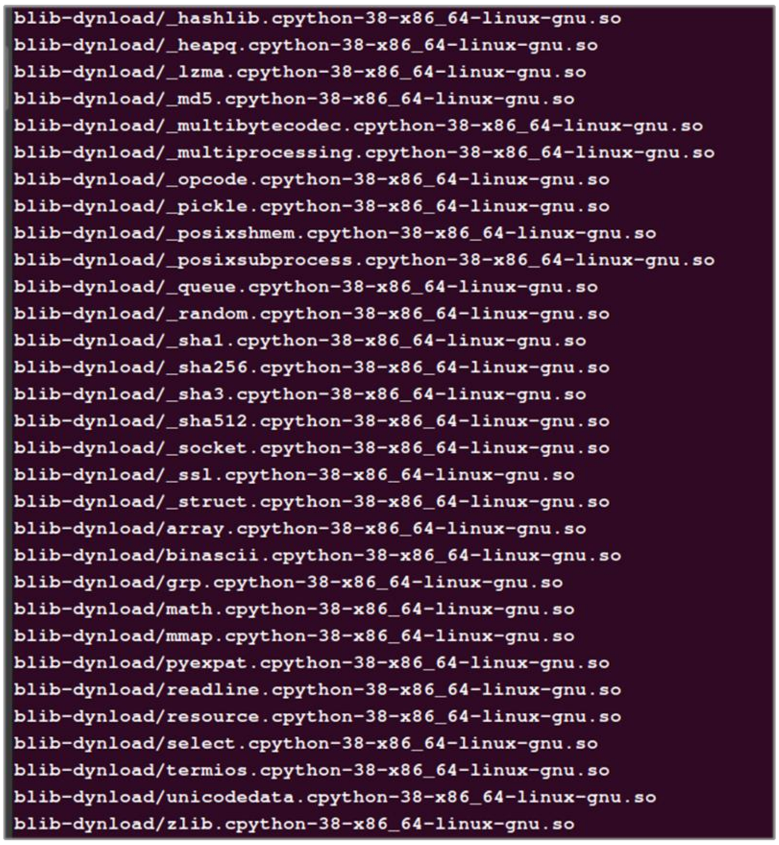
Flag: BOH21{w4tst0d0w1th0utU}

4 – NotSoDifficult

Description:

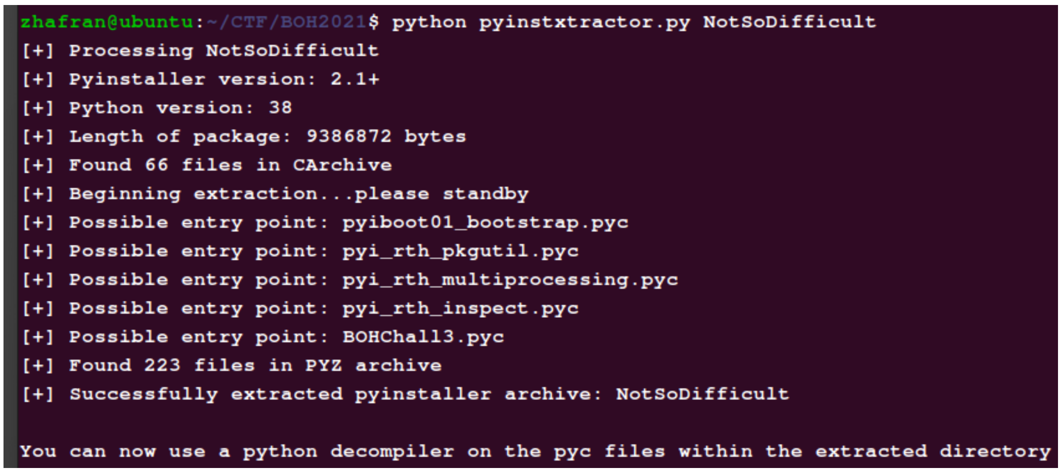
“Do not fear, this one will not be that difficult. But, no promise though. You just have to work smarter.”

If we print the strings of the binary among them will show a bunch of **“cpython”** words. This indicate that this binary is written in python. Which means that we can try to decompile the binary to get back the source code fully.



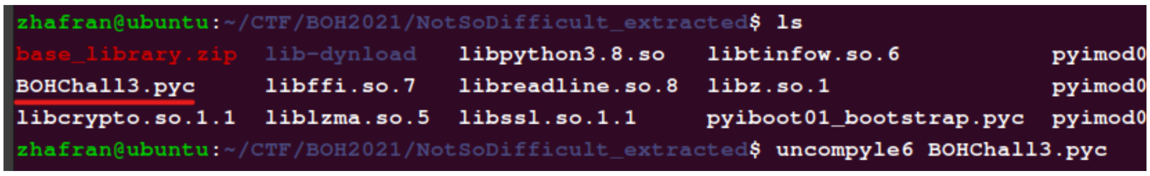
```
blib-dynload/_hashlib.cpython-38-x86_64-linux-gnu.so
blib-dynload/_heapq.cpython-38-x86_64-linux-gnu.so
blib-dynload/_lzma.cpython-38-x86_64-linux-gnu.so
blib-dynload/_md5.cpython-38-x86_64-linux-gnu.so
blib-dynload/_multibytecodec.cpython-38-x86_64-linux-gnu.so
blib-dynload/_multiprocessing.cpython-38-x86_64-linux-gnu.so
blib-dynload/_opcode.cpython-38-x86_64-linux-gnu.so
blib-dynload/_pickle.cpython-38-x86_64-linux-gnu.so
blib-dynload/_posixshmem.cpython-38-x86_64-linux-gnu.so
blib-dynload/_posixsubprocess.cpython-38-x86_64-linux-gnu.so
blib-dynload/_queue.cpython-38-x86_64-linux-gnu.so
blib-dynload/_random.cpython-38-x86_64-linux-gnu.so
blib-dynload/_sha1.cpython-38-x86_64-linux-gnu.so
blib-dynload/_sha256.cpython-38-x86_64-linux-gnu.so
blib-dynload/_sha3.cpython-38-x86_64-linux-gnu.so
blib-dynload/_sha512.cpython-38-x86_64-linux-gnu.so
blib-dynload/_socket.cpython-38-x86_64-linux-gnu.so
blib-dynload/_ssl.cpython-38-x86_64-linux-gnu.so
blib-dynload/_struct.cpython-38-x86_64-linux-gnu.so
blib-dynload/array.cpython-38-x86_64-linux-gnu.so
blib-dynload/binascii.cpython-38-x86_64-linux-gnu.so
blib-dynload/grp.cpython-38-x86_64-linux-gnu.so
blib-dynload/math.cpython-38-x86_64-linux-gnu.so
blib-dynload/mmap.cpython-38-x86_64-linux-gnu.so
blib-dynload/pyexpat.cpython-38-x86_64-linux-gnu.so
blib-dynload/readline.cpython-38-x86_64-linux-gnu.so
blib-dynload/resource.cpython-38-x86_64-linux-gnu.so
blib-dynload/select.cpython-38-x86_64-linux-gnu.so
blib-dynload/termios.cpython-38-x86_64-linux-gnu.so
blib-dynload/unicodedata.cpython-38-x86_64-linux-gnu.so
blib-dynload/zlib.cpython-38-x86_64-linux-gnu.so
```

Currently the binary is in the format of executable, we can use multiple python tools/Script to extract the pyc file. In this case we can use pyinstxtractor.



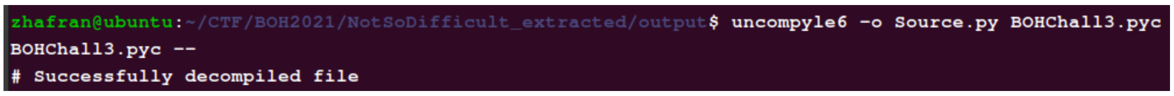
```
zhafran@ubuntu:~/CTF/BOH2021$ python pyinstxtractor.py NotSoDifficult
[+] Processing NotSoDifficult
[+] Pyinstaller version: 2.1+
[+] Python version: 38
[+] Length of package: 9386872 bytes
[+] Found 66 files in CArchive
[+] Beginning extraction...please standby
[+] Possible entry point: pyiboot01_bootstrap.pyc
[+] Possible entry point: pyi_rth_pkgutil.pyc
[+] Possible entry point: pyi_rth_multiprocessing.pyc
[+] Possible entry point: pyi_rth_inspect.pyc
[+] Possible entry point: BOHChall3.pyc
[+] Found 223 files in PYZ archive
[+] Successfully extracted pyinstaller archive: NotSoDifficult

You can now use a python decompiler on the pyc files within the extracted directory
```

Next, we can use a decompiler to get back the source code.

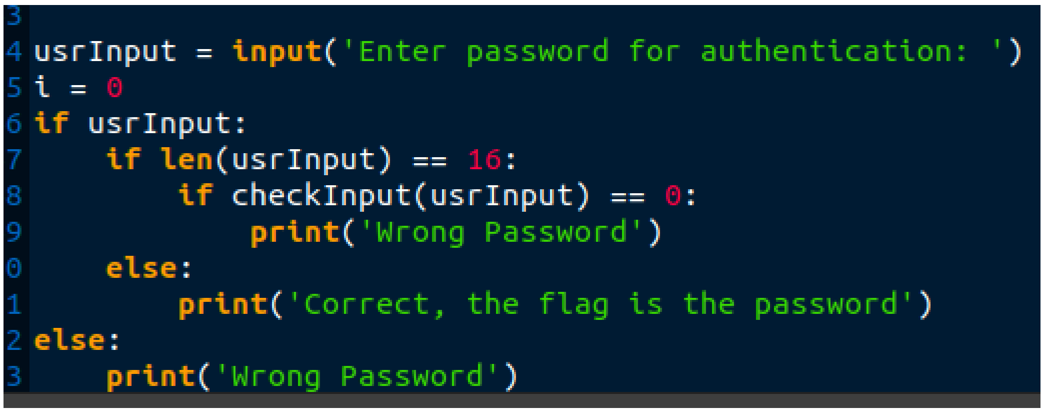


```
zhafran@ubuntu:~/CTF/BOH2021/NotSoDifficult_extracted$ ls
base_library.zip  lib-dynload  libpython3.8.so   libtinfow.so.6           pyimod0
BOHChall3.pyc     libffi.so.7  libreadline.so.8  libz.so.1                pyimod0
libcrypto.so.1.1  liblzma.so.5 libssl.so.1.1     pyiboot01_bootstrap.pyc  pyimod0
zhafran@ubuntu:~/CTF/BOH2021/NotSoDifficult_extracted$ uncompyle6 BOHChall3.pyc
```

As of this writing the most recent native Python decompiler is uncompyle6. Therefore, we can use the same decompiler to get the back the source code of the python. To make things easier we going to put down the output into "Source.py" so we can easily read the source code through a text editor.

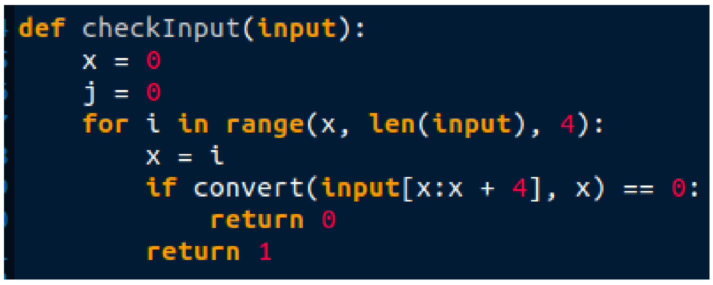


```
zhafran@ubuntu:~/CTF/BOH2021/NotSoDifficult_extracted/output$ uncompyle6 -o Source.py BOHChall3.pyc
BOHChall3.pyc --
# Successfully decompiled file
```



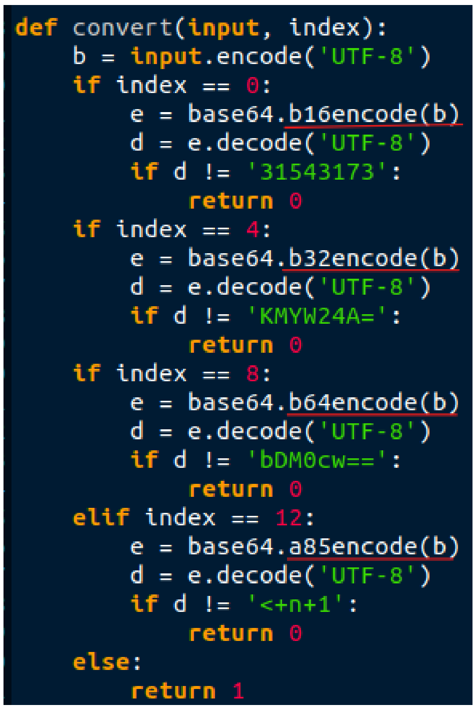
```python
usrInput = input('Enter password for authentication: ')
i = 0
if usrInput:
    if len(usrInput) == 16:
        if checkInput(usrInput) == 0:
            print('Wrong Password')
    else:
        print('Correct, the flag is the password')
else:
    print('Wrong Password')
```

main function



```python
def checkInput(input):
    x = 0
    j = 0
    for i in range(x, len(input), 4):
        x = i
        if convert(input[x:x + 4], x) == 0:
            return 0
        return 1
```

checkInput function



```python
def convert(input, index):
    b = input.encode('UTF-8')
    if index == 0:
        e = base64.b16encode(b)
        d = e.decode('UTF-8')
        if d != '31543173':
            return 0
    if index == 4:
        e = base64.b32encode(b)
        d = e.decode('UTF-8')
        if d != 'KMYW24A=':
            return 0
    if index == 8:
        e = base64.b64encode(b)
        d = e.decode('UTF-8')
        if d != 'bDM0cw==':
            return 0
    elif index == 12:
        e = base64.a85encode(b)
        d = e.decode('UTF-8')
        if d != '<+n+1':
            return 0
    else:
        return 1
```

convert function

At this point things should be straightforward. From the main function it will check the length of our input with 16. **Now we know the correct input is 16 character**. Next, the program will take the characters of the input 4 at a time and convert it with a pre-initialized string that is in the format of different bases. **We can solve this challenge by taking each of the strings and decode it based on their encoding base. (base16, base32, base64, base85).**

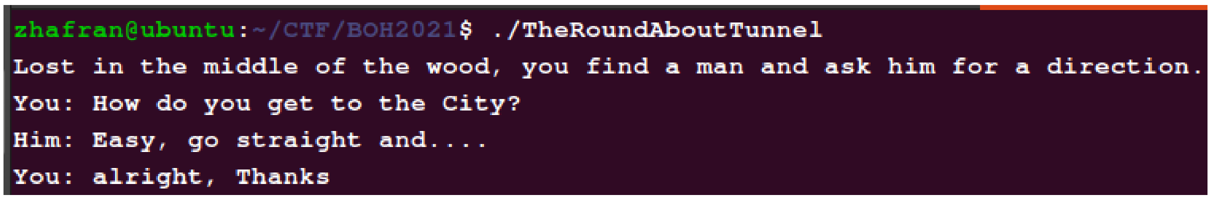
Flag: BOH21{1T1sS1mpl34sTh4t}

5 – TheRoundAboutTunnel

Description:

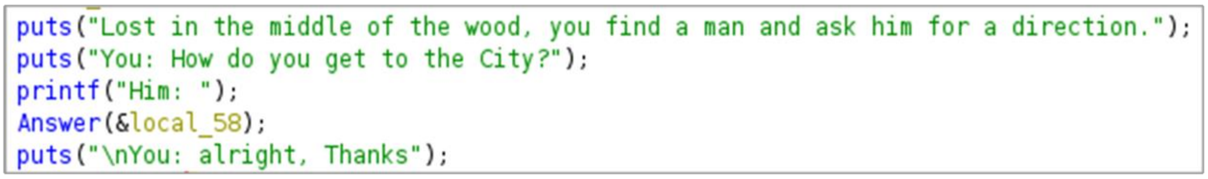
“People of Today Think Deeply Instead of Clearly. Perhaps the answer is right in front of us.”

Running the binary give us an output of some conversation between two people.



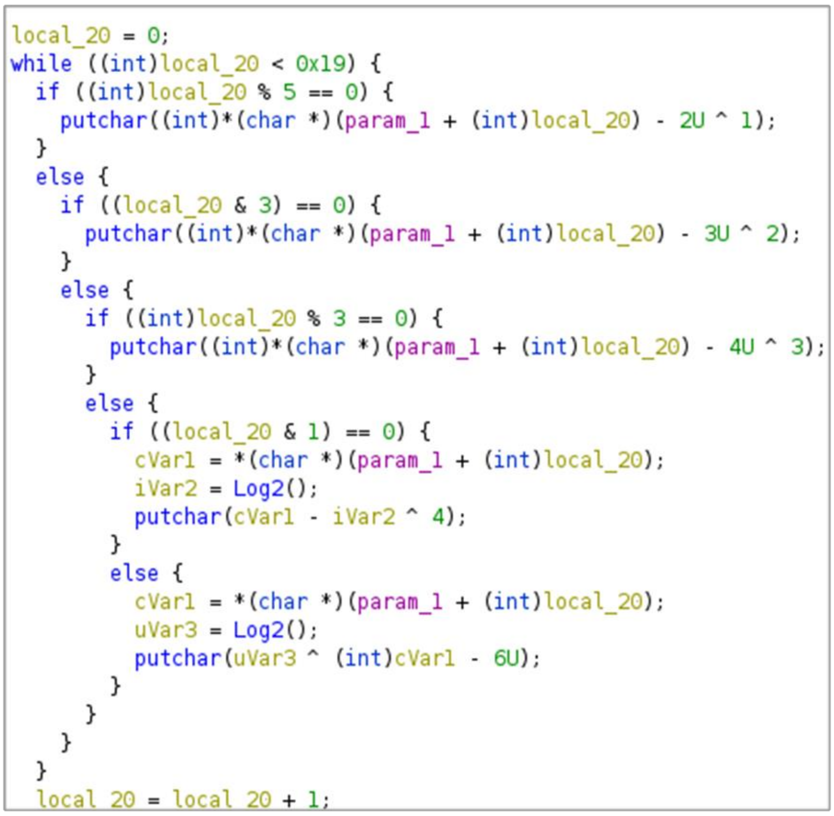
```
zhafran@ubuntu:~/CTF/BOH2021$ ./TheRoundAboutTunnel
Lost in the middle of the wood, you find a man and ask him for a direction.
You: How do you get to the City?
Him: Easy, go straight and....
You: alright, Thanks
```

Opening it on Ghidra we can see that it calls a function called **Answer** and use a variable called local_58 as their argument.



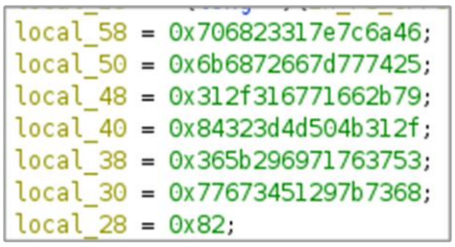
```
puts("Lost in the middle of the wood, you find a man and ask him for a direction.");
puts("You: How do you get to the City?");
printf("Him: ");
Answer(&local_58);
puts("\nYou: alright, Thanks");
```

Inside the Answer function we can see that it will do a while loop while it will print a character depending on the condition inside the if statement. We can see that the condition of While loop is set to less than 0x19.



```
local_20 = 0;
while ((int)local_20 < 0x19) {
  if ((int)local_20 % 5 == 0) {
    putchar((int)*(char *)(param_1 + (int)local_20) - 2U ^ 1);
  }
  else {
    if ((local_20 & 3) == 0) {
      putchar((int)*(char *)(param_1 + (int)local_20) - 3U ^ 2);
    }
    else {
      if ((int)local_20 % 3 == 0) {
        putchar((int)*(char *)(param_1 + (int)local_20) - 4U ^ 3);
      }
      else {
        if ((local_20 & 1) == 0) {
          cVar1 = *(char *)(param_1 + (int)local_20);
          iVar2 = Log2();
          putchar(cVar1 - iVar2 ^ 4);
        }
        else {
          cVar1 = *(char *)(param_1 + (int)local_20);
          uVar3 = Log2();
          putchar(uVar3 ^ (int)cVar1 - 6U);
        }
      }
    }
  }
  local_20 = local_20 + 1;
```

However, we can see that the input parameter length (local_58 length) is more than 0x19 Which means it might not be printing all of the possible characters.



```
local_58 = 0x706823317e7c6a46;
local_50 = 0x6b6872667d777425;
local_48 = 0x312f316771662b79;
local_40 = 0x84323d4d504b312f;
local_38 = 0x365b296971763753;
local_30 = 0x77673451297b7368;
local_28 = 0x82;
```

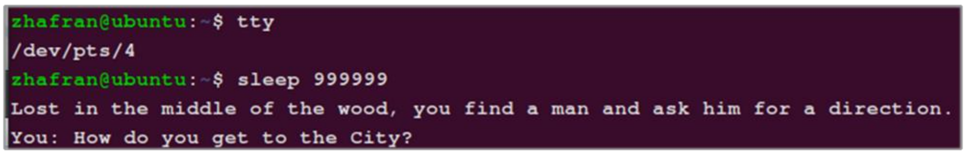
Obfuscated Strings

To solve this challenge, we can either take the obfuscated strings and create some Script to deobfuscate it or we can patch / dynamically run the program and change the condition of the while loops.

If we want to deobfuscate the strings manually we can see that the program will do a nested if statement and check the current index of the strings. It will in order check if the index is the multiple of 5, 4, 3, 2, or 1. Depending on that the value of which it will be XORed will be different. On top of that in some cases it will call a Log2 function and will use the output of that as the XORed number. The Log2 function itself is just to do Logarithm by 2 to the input parameter.

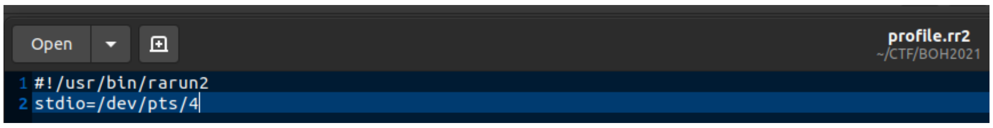
For the other way we can use Radare2 or any other debugger to do the job. To make it easier we going to set up a second terminal to get the stdout output of our binary program.

On the second terminal we use "**tty**" command to get the name of the terminal. We then need to put it to sleep so that it will not interfere with the program output.



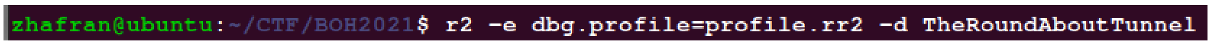
```
zhafran@ubuntu:~$ tty
/dev/pts/4
zhafran@ubuntu:~$ sleep 999999
Lost in the middle of the wood, you find a man and ask him for a direction.
You: How do you get to the City?
```

Then we going to create **"rarun"** (RadareRun) script and put the name of our terminal there.



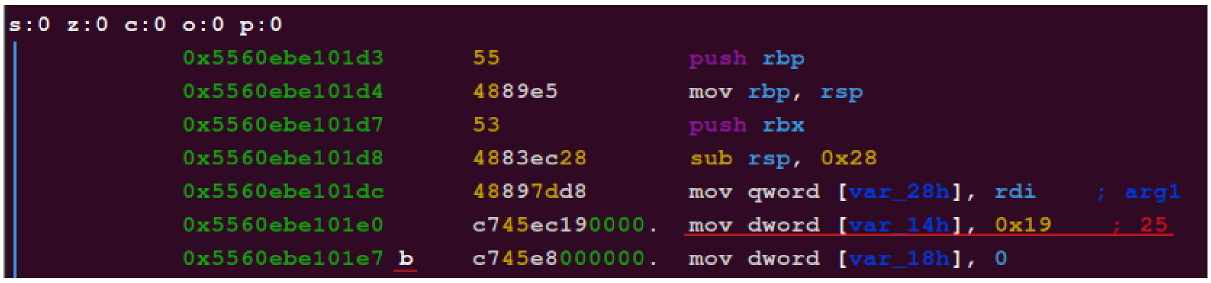
```
#!/usr/bin/rarun2
stdio=/dev/pts/4
```

Then we can just run the binary using extra -e argument and specify our rr2 script file.



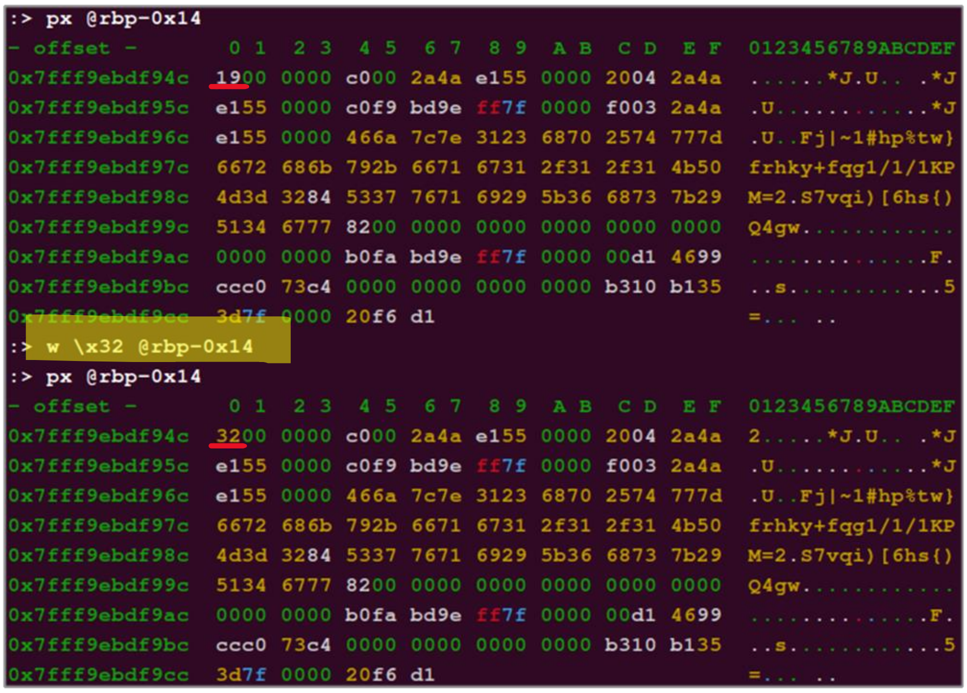
```
zhafran@ubuntu:~/CTF/BOH2021$ r2 -e dbg.profile=profile.rr2 -d TheRoundAboutTunnel
```

Next, we just have to change the value of the loop variable limit, in this case the name is **var_14h**. We need to set a breakpoint right after the variable value has been initialized.



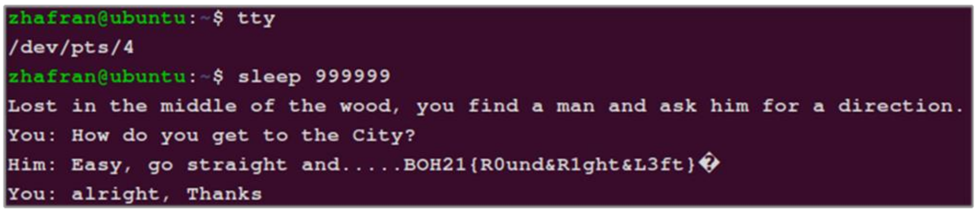
```
s:0 z:0 c:0 o:0 p:0
            0x5560ebe101d3      55              push rbp
            0x5560ebe101d4      4889e5          mov rbp, rsp
            0x5560ebe101d7      53              push rbx
            0x5560ebe101d8      4883ec28        sub rsp, 0x28
            0x5560ebe101dc      48897dd8        mov qword [var_28h], rdi    ; arg1
            0x5560ebe101e0      c745ec190000.   mov dword [var_14h], 0x19    ; 25
            0x5560ebe101e7 b    c745e8000000.   mov dword [var_18h], 0
```

As you can see if we print the register it will show 0x19. Then we use **write** command to change the register value to 0x32. Then we print the register again we can see that now it is 0x32. After this we just need to let the program run until the end.



```
:> px @rbp-0x14
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7fff9ebdf94c  1900 0000 c000 2a4a e155 0000 2004 2a4a  ......*J.U.. .*J
0x7fff9ebdf95c  e155 0000 c0f9 bd9e ff7f 0000 f003 2a4a  .U............*J
0x7fff9ebdf96c  e155 0000 466a 7c7e 3123 6870 2574 777d  .U..Fj|~1#hp%tw}
0x7fff9ebdf97c  6672 686b 792b 6671 6731 2f31 2f31 4b50  frhky+fqg1/1/1KP
0x7fff9ebdf98c  4d3d 3284 5337 7671 6929 5b36 6873 7b29  M=2.S7vqi)[6hs{)
0x7fff9ebdf99c  5134 6777 8200 0000 0000 0000 0000 0000  Q4gw............
0x7fff9ebdf9ac  0000 0000 b0fa bd9e ff7f 0000 00d1 4699  ..............F.
0x7fff9ebdf9bc  ccc0 73c4 0000 0000 0000 0000 b310 b135  ..s............5
0x7fff9ebdf9cc  3d7f 0000 20f6 d1                        =... ..
:> w \x32 @rbp-0x14
:> px @rbp-0x14
- offset -       0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0123456789ABCDEF
0x7fff9ebdf94c  3200 0000 c000 2a4a e155 0000 2004 2a4a  2.....*J.U.. .*J
0x7fff9ebdf95c  e155 0000 c0f9 bd9e ff7f 0000 f003 2a4a  .U............*J
0x7fff9ebdf96c  e155 0000 466a 7c7e 3123 6870 2574 777d  .U..Fj|~1#hp%tw}
0x7fff9ebdf97c  6672 686b 792b 6671 6731 2f31 2f31 4b50  frhky+fqg1/1/1KP
0x7fff9ebdf98c  4d3d 3284 5337 7671 6929 5b36 6873 7b29  M=2.S7vqi)[6hs{)
0x7fff9ebdf99c  5134 6777 8200 0000 0000 0000 0000 0000  Q4gw............
0x7fff9ebdf9ac  0000 0000 b0fa bd9e ff7f 0000 00d1 4699  ..............F.
0x7fff9ebdf9bc  ccc0 73c4 0000 0000 0000 0000 b310 b135  ..s............5
0x7fff9ebdf9cc  3d7f 0000 20f6 d1                        =... ..
```

After running the program, we can see that on our second terminal we will get the stdout output.



```
zhafran@ubuntu:~$ tty
/dev/pts/4
zhafran@ubuntu:~$ sleep 999999
Lost in the middle of the wood, you find a man and ask him for a direction.
You: How do you get to the City?
Him: Easy, go straight and.....BOH21{R0und&R1ght&L3ft}�
You: alright, Thanks
```

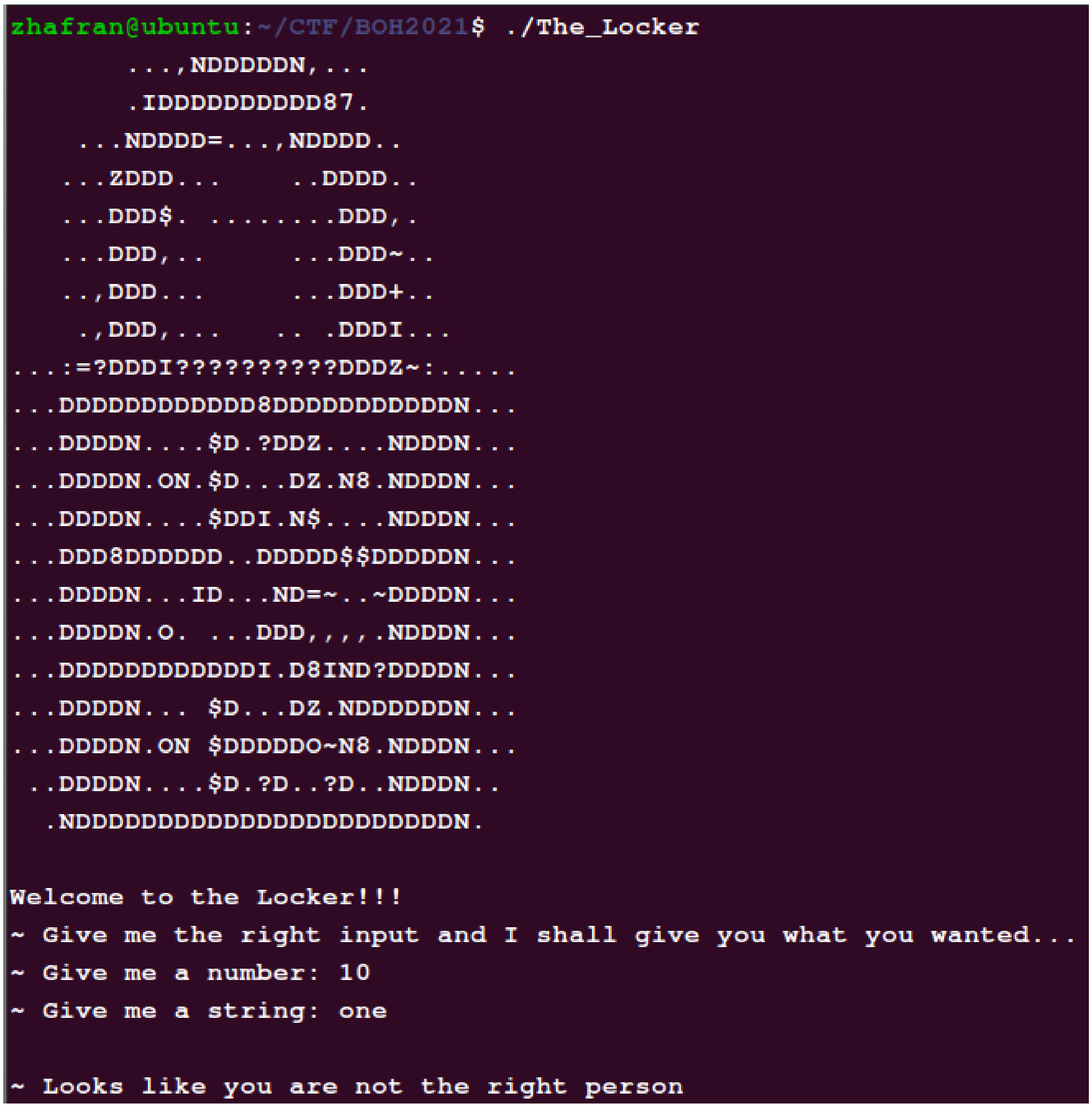
Flag: BOH21{R0und&R1ght&L3ft}

6 – The Locker

Description:

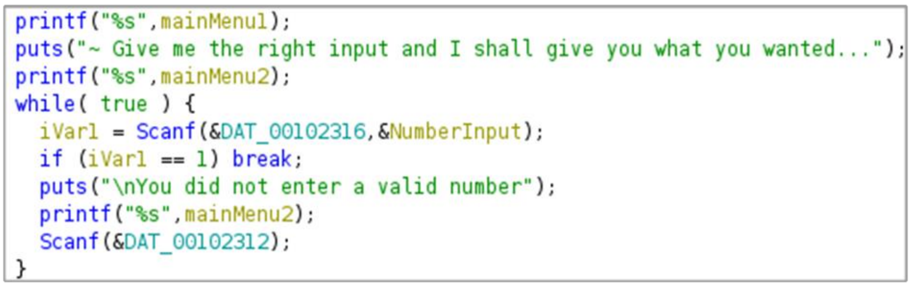
“The rise of the Virtual Locker has gaining attention over people from the internet. Legend said opening the locker will give the unlocker something immeasurable in value.”

Running the program will prompt us for a number. After we give a number it will then ask for a String. Then it will just give us a fail output.



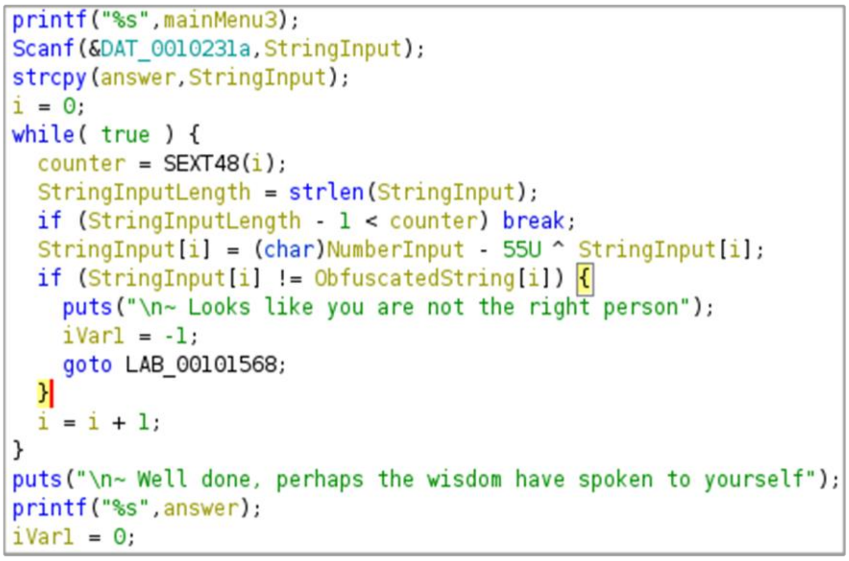
```
zhafran@ubuntu:~/CTF/BOH2021$ ./The_Locker
       ...,NDDDDDN,...
       .IDDDDDDDDDD87.
    ...NDDDD=...,NDDDD..
   ...ZDDD...    ..DDDD..
   ...DDD$. ........DDD,.
   ...DDD,..     ...DDD~..
   ..,DDD...     ...DDD+..
    .,DDD,...   .. .DDDI...
...:=?DDDI??????????DDDZ~:.....
...DDDDDDDDDDDD8DDDDDDDDDDN...
...DDDDN....$D.?DDZ....NDDDN...
...DDDDN.ON.$D...DZ.N8.NDDDN...
...DDDDN....$DDI.N$....NDDDN...
...DDD8DDDDDD..DDDDD$$DDDDDN...
...DDDDN...ID...ND=~..~DDDDN...
...DDDDN.O. ...DDD,,,,.NDDDN...
...DDDDDDDDDDDI.D8IND?DDDDN...
...DDDDN... $D...DZ.NDDDDDDN...
...DDDDN.ON $DDDDDO~N8.NDDDN...
 ..DDDDN....$D.?D..?D..NDDDN..
  .NDDDDDDDDDDDDDDDDDDDDDDDDN.

Welcome to the Locker!!!
~ Give me the right input and I shall give you what you wanted...
~ Give me a number: 10
~ Give me a string: one

~ Looks like you are not the right person
```

Opening it on Ghidra we will see that the program will print a String that is used for main menu. It will then ask us to enter a number and will store the value in variable called **NumberInput**. We can see that in case we did not enter a valid number the program will keep on looping until we entered a valid one.



```
printf("%s",mainMenu1);
puts("~ Give me the right input and I shall give you what you wanted...");
printf("%s",mainMenu2);
while( true ) {
  iVar1 = Scanf(&DAT_00102316,&NumberInput);
  if (iVar1 == 1) break;
  puts("\nYou did not enter a valid number");
  printf("%s",mainMenu2);
  Scanf(&DAT_00102312);
}
```

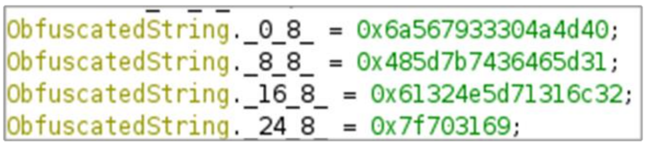
The next part of the program will ask us for a strings and then it will create a loop where it will **XOR our input with the number that we use as an input before subtracted by 55 decimal**. Our goal now is to find out what value must be XORed with our input so that it will match with our Obfuscated Strings.



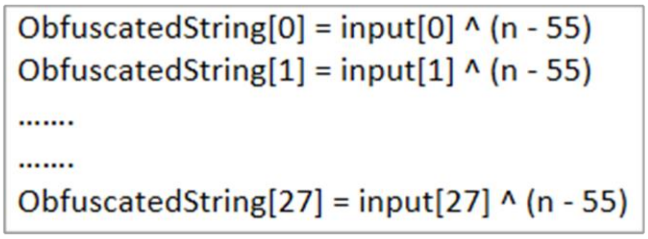
```
printf("%s",mainMenu3);
Scanf(&DAT_0010231a,StringInput);
strcpy(answer,StringInput);
i = 0;
while( true ) {
  counter = SEXT48(i);
  StringInputLength = strlen(StringInput);
  if (StringInputLength - 1 < counter) break;
  StringInput[i] = (char)NumberInput - 55U ^ StringInput[i];
  if (StringInput[i] != ObfuscatedString[i]) {
    puts("\n~ Looks like you are not the right person");
    iVar1 = -1;
    goto LAB_00101568;
  }
  i = i + 1;
}
puts("\n~ Well done, perhaps the wisdom have spoken to yourself");
printf("%s",answer);
iVar1 = 0;
```

Through Ghidra we can see what our Obfuscated Strings is. Since it is in Little Endian we need to reverse it.

404d4a303379566a315d4636747b5d48326c31715d4e32616931707f



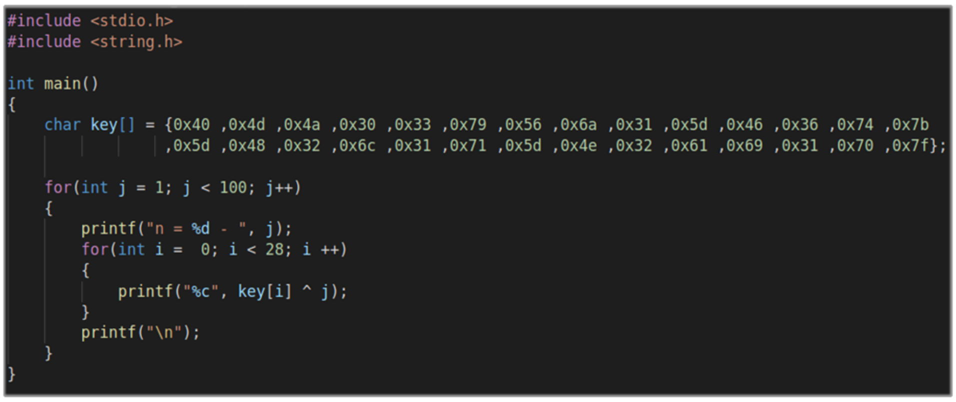
```
ObfuscatedString._0_8_  = 0x6a567933304a4d40;
ObfuscatedString._8_8_  = 0x485d7b7436465d31;
ObfuscatedString._16_8_  = 0x61324e5d71316c32;
ObfuscatedString._24_8_  = 0x7f703169;
```



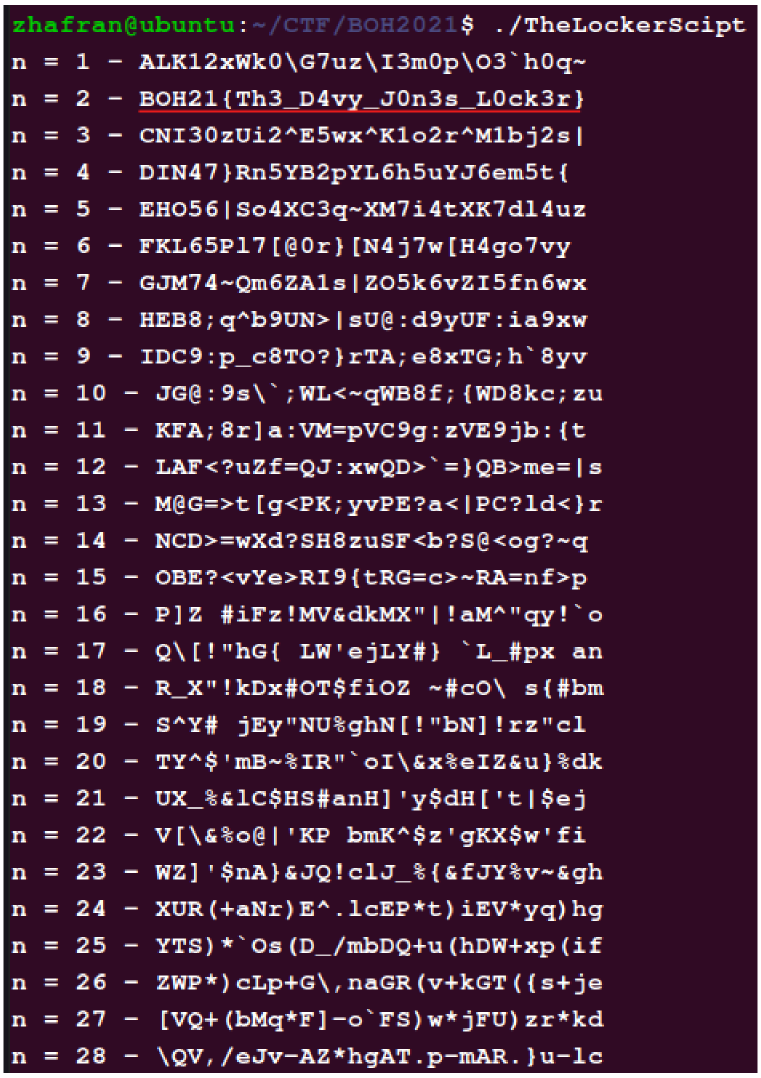
ObfuscatedString[0] = input[0] ^ (n - 55)
ObfuscatedString[1] = input[1] ^ (n - 55)
....
....
ObfuscatedString[27] = input[27] ^ (n - 55)

The equation

Although the equation above is mathematically impossible to solve since there are two values that we do not know the **input** and **n**. In reality we can just create a simple script that can brute force the answer.



```c
#include <stdio.h>
#include <string.h>

int main()
{
    char key[] = {0x40 ,0x4d ,0x4a ,0x30 ,0x33 ,0x79 ,0x56 ,0x6a ,0x31 ,0x5d ,0x46 ,0x36 ,0x74 ,0x7b
                 ,0x5d ,0x48 ,0x32 ,0x6c ,0x31 ,0x71 ,0x5d ,0x4e ,0x32 ,0x61 ,0x69 ,0x31 ,0x70 ,0x7f};

    for(int j = 1; j < 100; j++)
    {
        printf("n = %d - ", j);
        for(int i =  0; i < 28; i ++)
        {
            printf("%c", key[i] ^ j);
        }
        printf("\n");
    }
}
```



```
zhafran@ubuntu:~/CTF/BOH2021$ ./TheLockerScipt
n = 1 - ALK12xWk0\G7uz\I3m0p\O3`h0q~
n = 2 - BOH21{Th3_D4vy_J0n3s_L0ck3r}
n = 3 - CNI30zUi2^E5wx^K1o2r^M1bj2s|
n = 4 - DIN47}Rn5YB2pYL6h5uYJ6em5t{
n = 5 - EHO56|So4XC3q~XM7i4tXK7dl4uz
n = 6 - FKL65Pl7[@0r}[N4j7w[H4go7vy
n = 7 - GJM74~Qm6ZA1s|ZO5k6vZI5fn6wx
n = 8 - HEB8;q^b9UN>|sU@:d9yUF:ia9xw
n = 9 - IDC9:p_c8TO?}rTA;e8xTG;h`8yv
n = 10 - JG@:9s\`;WL<~qWB8f;{WD8kc;zu
n = 11 - KFA;8r]a:VM=pVC9g:zVE9jb:{t
n = 12 - LAF<?uZf=QJ:xwQD>`=}QB>me=|s
n = 13 - M@G=>t[g<PK;yvPE?a<|PC?ld<}r
n = 14 - NCD>=wXd?SH8zuSF<b?S@<og?~q
n = 15 - OBE?<vYe>RI9{tRG=c>~RA=nf>p
n = 16 - P]Z #iFz!MV&dkMX"|!aM^"qy!`o
n = 17 - Q\[!"hG{ LW'ejLY#} `L_#px an
n = 18 - R_X"!kDx#OT$fiOZ ~#cO\ s{#bm
n = 19 - S^Y# jEy"NU%ghN[!"bN]!rz"cl
n = 20 - TY^$'mB~%IR"`oI\&x%eIZ&u}%dk
n = 21 - UX_%&lC$HS#anH]'y$dH['t|$ej
n = 22 - V[\&%o@|'KP bmK^$z'gKX$w'fi
n = 23 - WZ]'$nA}&JQ!clJ_%{&fJY%v~&gh
n = 24 - XUR(+aNr)E^.lcEP*t)iEV*yq)hg
n = 25 - YTS)*`Os(D_/mbDQ+u(hDW+xp(if
n = 26 - ZWP*)cLp+G\,naGR(v+kGT({s+je
n = 27 - [VQ+(bMq*F]-o`FS)w*jFU)zr*kd
n = 28 - \QV,/eJv-AZ*hgAT.p-mAR.}u-lc
```

Flag: BOH21{Th3_D4vy_J0n3s_L0ck3r}