DESCRIPTION

The index check against array boundaries is very peculiar.

Connect to 141.85.224.106:31348 and get the flag.

RESOURCES

As part of the challenge I received an executable file called **fibonacci** as an attachment for analysis as well as a **libc** version used for compilation.

APPROACHES

1. The first approach here was to run the program and after running it I could see that it is requiring an input to obtain the **fibbonacci** number at that position. As well, here, another interesting observation is that for small enough numbers, the result is returned and also another input is awaited. However, for big enough numbers, the program ends immediatly.

```
mihnea@HOME-PC:/mnt/c/Users/mblot/Desktop/CNS/Tema3_Moodle/fibonacci$ ./fibonacci
What fibonacci number do you want?
5
8
What fibonacci number do you want?
1000
```

2. Thus, the decompilation of the **run function** looks like this:

```
{
  char local_28 [28];
  uint local_c;

  make_fibo();
  while( true ) {
    puts("What fibonacci number do you want?");
    gets(local_28);
    local_c = atoi(local_28);
    if (0x2e < (int)(((int)local_c >> 0x1f ^ local_c) - ((int)local_c >> 0x1f))) break;
    printf("%lu\n",(&fibonacci)[(int)local_c]);
  }
  return;
}
```

3. Here we can see that a **make_fibo** function is called and then, we have an unsecure **gets** function called (we are going to use it for a buffer overflow). Then, we have an **atoi** called on the given input (which from manual transforms to **integer** the first part of the string that can be transformed and nothing more). This means, that we will make sure the first part of the string is a number (big enough to match the following if and then **break** from the **while true** loop.

4. So, the idea here is to send a big number such that the next **if** condition is sattisfied and the **while true** loop gets stopped such that we will imeddiatly return after it. I chose „1” * 8 as a big number as we are on **64 bit architecture**.

5. The buffer in which we are reading is **0x20** bytes big according to decompilation:

```
4006cd:      48 8d 45 e0             lea    rax,[rbp-0x20]
4006d1:      48 89 c7                mov    rdi,rax
4006d4:      b8 00 00 00 00          mov    eax,0x0
4006d9:      e8 52 fe ff ff          call   400530 <gets@plt>
```

6. Then, for the next (**0x20** – **8** (the size of the initial number) + **8** (the old_rbp_size)) we can give garbage input values such that we just end at overriding the return address of **the run function**. We override the return addreess to a **pop_rdi_ret** gadget that will put the first parameter as **puts@got** and then will call **puts@plt** with it such that we leak the address of **puts** in our version of **libc** and then having the **offset** of **puts** we can determine the start of **libc** and any function inside it. We are interested of **system(„/bin/sh”)**.

7. Then, after leaking the **puts** address we will jump back to beginning of **main** function where we will override the return address of **run function** again to a **pop_rdi_ret** gadget that will put the first parameter as the **„/bin/sh”** string and the will call **system** with it.

8. If you run the script (**python3 script.py**), you will get a shell on the server and if we run **cat /home/ctf/flag** we will get the flag which is:

CNS_CTF{2923b48b850742e1ec7c096e3d617cb5}