DESCRIPTION

Connect to the service at 141.85.224.106:31336. It will give you a friendly "Hello", and something more...

RESOURCES

As part of the challenge I received an executable file called hidden as an attachment for analysis.

APPROACHES

1. The first thing to do here was to run the program and I saw that an user input is required and the format returned is **Hello, <user_input>**. Additionally, I ran it for multiple inputs and I observed that for input strings longer than a specific length, apart from the previous menitoned output, it also returns another character. I have always seen just 1.

2. Then, I decompiled it with Ghidra to get a better understanding of what is happening:

```
read_flag(auStack_44,param_2,pcVar4 + 6);
local_10 = (char *)malloc(0x200);
if (local_10 == (char *)0x0) {
  uVar1 = 0xffffffff;
}
else {
  memcpy(local_58,"Hello, ",7);
  fgets(local_10,0x46,stdin);
  sVar2 = strlen(local_10);
  memcpy(local_58 + 7,local_10,sVar2);
  sVar2 = strlen(local_10);
  auStack_50[sVar2] = 0;
  puts(local_58);
  uVar1 = 0;
}
return uVar1;
```

3. This is what I got from Ghidra and from what I could see there was a call for **read_flag**, and then a construction of a string where we put „Hello, ” and then we concatenate what we read form the keyboard and then this string gets printed on the screen. Additionally, considering the additional chacarter that go printed on the screen while experimenting the binary, I thought that it has to be from the same printing as I could not find any other print command.

4. Thus, I went in the assembly code to see better how these functions are called:

```
4008b0:      48 8d 45 b0             lea    rax,[rbp-0x50]
4008b4:      48 83 c0 14             add    rax,0x14
4008b8:      48 89 c7                mov    rdi,rax
4008bb:      e8 26 ff ff ff          call   4007e6 <read_flag>
```

5. So, as we can see here, **read_flag** is called with a buffer at the address [rbp -0x50 + 0x14] = [rbp – 0x3C]

```
4008dc:       48 8d 45 b0              lea    rax,[rbp-0x50]
4008e0:       ba 07 00 00 00           mov    edx,0x7
4008e5:       be f8 09 40 00           mov    esi,0x4009f8
4008ea:       48 89 c7                 mov    rdi,rax
4008ed:       e8 ae fd ff ff           call   4006a0 <memcpy@plt>
```

6. Here we can see that the first **memcpy** is using a buffer at the address [rbp – 0x50] which means that the **memcpy** that is putting „Hello, ” and the **read_flag** functions are manipulating the same buffer.

```
400919:       48 8d 45 b0              lea    rax,[rbp-0x50]
40091d:       48 83 c0 07              add    rax,0x7
400921:       48 8b 4d f8              mov    rcx,QWORD PTR [rbp-0x8]
400925:       48 89 ce                 mov    rsi,rcx
400928:       48 89 c7                 mov    rdi,rax
40092b:       e8 70 fd ff ff           call   4006a0 <memcpy@plt>
```

7. Also, the last **memcpy** is using a buffer at [rbp – 0x50 + 0x7] which means [rbp – 0x49]

```
400937:       e8 24 fd ff ff           call   400660 <strlen@plt>
40093c:       48 83 c0 08              add    rax,0x8
400940:       c6 44 05 b0 00           mov    BYTE PTR [rbp+rax*1-0x50],0x0
```

8. At the end we are computing the lenght of the „Hello, <user_input>” we are adding 8 and then at that position we are putting a NULL character.

9. That being said it means that we are starting the buffer with 7 characters from „Hello, ” (rbp – 0x50 -> rbp – 0x49 = 7 bytes) and then a user input string from (rbp - 0x48 -> rbp - 0x3c = 12 bytes) and then the flag is read by the **read_flag** function. But we are also putting a NULL character at [rbp – 0x50 + length] so that means that if we fill al the 7 bytes from hello and all the 12 bytes from they user given input, we are going to have a string of length 19, but we are adding 0x8 to the length so that means that we are going to put the NULL value at position 20 which means that we are always going to see 1 character from the flag.

10. So, if we are going to send a 12 bytes user input we are going to see the first character, then with at 13 bytes user input we are going to see the second character and so on. So my idea in the script was to run the app multiple times and each time with a input string larger with one character starting from 12 until the end of the flag. Each time I would gather the character returned and add it to the final flag.

```
{
  int __fd;
  int local_c;

  __fd = open("/home/ctf/flag",0);
  if (__fd < 0) {
    perror("open");
                    /* WARNING: Subroutine does not return */
    exit(1);
  }
  read(__fd,param_1,0x32);
  for (local_c = 0; local_c < 0x32; local_c = local_c + 1) {
    *(byte *)((long)param_1 + (long)local_c) =
         (byte)local_c ^ *(byte *)((long)param_1 + (long)local_c);
  }
  close(__fd);
  return;
}
```

11. Here we also have to take care that the **read_flag** function also does a XOR operation for each character with it's position so when seeing the flag we have to do a XOR again with its position to obtain the initial character. Thus, I also implemented this in the script.

12. So, if you run the script (**python3 script.py**) and wait for a few seconds to send all the inputs and receive the answer from the server, you will obtain the final flag which is:

CNS_CTF{7ed0abc42c396e9b76d1928511886e08}