Always go further

Analysis

A `god_mode` function is present and opens a shell. It looks like the one that we have to call. However, since ASLR and PIE are present, we cannot know in advance its address.

We'll need to leak one to calculate the offset. Fortunately, it is exactly what the function `find_my_way` does ! If we can call this function, we would be able to calculate `god_mod` address.

This function can be called if the player is an *EXPLORER*. However, it seems they are no way to get an *EXPLORER* in this game. We would like that the player->role is equal to 2. So, we need to overflow `name` to write the value we want in player->role. If we can do that, we'll just have to choose *Do character action* and then leak the address.

We can see that name is written in `init_player` and uses a `max_size` variable to prevent overflow. This max_size is defined by `name_size` in main. But this variable is protected by a canary, so even if we can overflow a buffer, we need to know the canary.

This canary is harcoded and is always the saknow it is located on the stame. We have to find a way to read it. We ck so we need to read it. One option is format string attack, so let's find if a format string vulnerability is present.

If the player doesn't precise a name, the program will display a *usage* message. However, this one gets the program name and then put it in the printf function. If the program name is a format string, it will be interpreted ! We now have a way to read the canary.

Let's go back to `name_size`. We need to find an overflow somewhere. If the debug mode is activated with the -D param, we write a message in the `launch` buffer. This buffer is **LAUNCH_LENGTH** long and this constant is used in snprintf. However, the buffer has already been filled and we start writing 8 characters after the beginning. So we can have 8 byte overflow, which is enough to overwrite `name_size`. A ROP or ret2libc attack wouldn't work due to this small overflow.

Now that name_size is overwritten, we can overflow our player name and get the *EXPLORER* role to leak an address.

However, we just have the `god_mode` address but we can't call it. We cannot use the return address so we have to find another way to call the `god_mod` function in main body. When player performs an action, a function pointer is used. This one will call the function at the address it points. This is our only way to call the function we want. However, this function pointer is located before the name in the structure so we can't overlow it. In all cases, we would have to know the address before launching the program.

After looking at the player_t struct and npc_t struct, we see that they have the same size. More, npc game id is at the same location as player action, and we can change npc game id ! If we have a way that our player pointer points to the npc and then call perform action, the function at the address *proposed_game* will be called.

When we kill our player, only the role is set to *DEAD* and the pointer isn't reseted. It means we can perform a use after free or a double free attack. Use after isn't possible due to the role *DEAD* which is given because of the npc name with zeros at the end.

However, by killing twice our player, then creating a new one and then a npc, npc data will overwrite player data and the player pointer will point on it. We then only have to change the `proposed_game` to the address of `god_mode` and perform player action. We'll get our shell.

Exploitation

First, lets leak the canary. We can create a symlink to change the file name. After some tests we notice that for `./%20$p`, we get *0x14* which is name_size. So we can get the canary with `./%19$p`. We get `0xb4db16d` !

Now, we need to overwrite `name_size` and set the role to 2. One more time we have to change the file name. For a weird reason, we only need 30 characters of padding (not the 32 which are 40 - 8). The filename becomes:

`echo -n -e "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x6d\xb1\x4d\x0b0000"`

and the name is :

`echo -n -e "AAAAAAAAAAAAAAAAAAAA\x02"`

So we have the following command to execute :

`./$(echo -n -e "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\x6d\xb1\x4d\x0b0000") $(echo -n -e "AAAAAAAAAAAAAAAAAAAA\x02") -D`

Let's create a new farmer with **1**, then perform the action with **2**.

```
You're located at 0x5658c2ae
```

We got the `find_my_way` address ! We need to calculate the offset beetween `find_my_way` and `god_mode`. Let's use `objdump`: `objdump -d challenge`

```
0000125d <god_mode>:
    125d:       55                      push   %ebp
    125e:       89 e5                   mov    %esp,%ebp

...
...

000012ae <find_my_way>:
    12ae:       55                      push   %ebp
```

The difference is about 0x12ae - 0x125d. The god_mod address is so 0x5658c2ae - (0x12ae - 0x125d) = 1448657501.

Kill twice the player

```
Your choice: 6
This life is too hard for... Bye bye AAAAAAAAAAAAAAAAAAAA
...
...
Your choice: 6
This life is too hard for... Bye bye Cy�
```

Now, create a new player and then a npc with **1** and **3**.

Change the game to *1448657501*

```
Your choice: 5
Choose a game number (1-2):
1.      dice
2.      more or less
1448657501
You've chosen game n°1448657501 !What do you want to do ?
```

And we can perform our player action to get the shell !

```
Your choice: 2
sh-5.1$
```