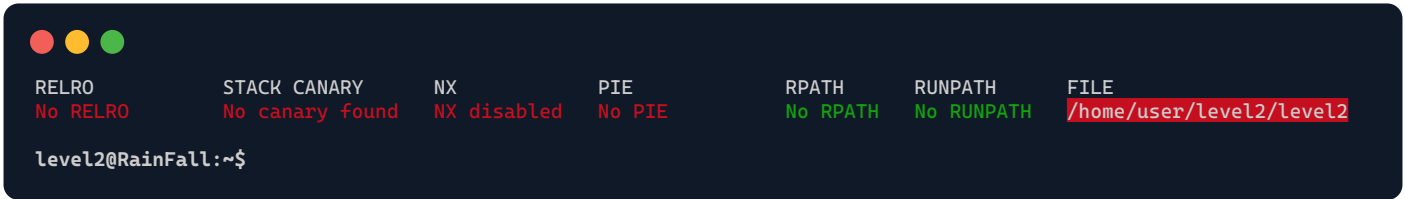


./level2



Decompiled file with *Ghidra*:



The program is designed to process user input, then check the top bits of its *return address*. When it identifies the **0xb...** pattern, common to *stack addresses* in systems such as **Linux**, it immediately terminates. This is a built-in security measure to counteract attempts to inject *shellcode* into the **stack**.

Attack Vectors:

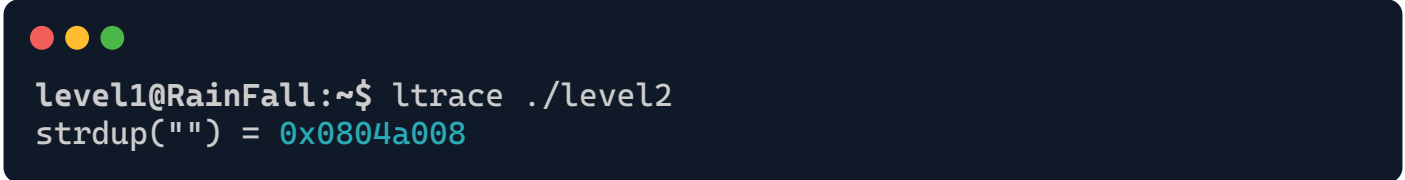
- The use of **gets(userInput)** is a notable weak point. It's susceptible to *buffer overflows*, allowing us to manipulate the **stack**, including the function's *return address*, like in the last level.
- The function **strdup(userInput)** duplicates the input but doesn't manage the memory afterward, leading to a *memory leak*. In certain scenarios, this can be turned into an *exploit*.

Given that our program doesn't provide direct command execution methods like **system** or **execve**, we'd lean towards using **shellcode**, a compact code designed for *software exploitation*, which would let us launch a **shell**.

Although the program checks and prevents return addresses that point to the **stack** (those starting with 0xb...), it doesn't stop us from changing it to a **heap** address.

So, what's our move? Leveraging the memory leak caused by **strdup** looks promising.

To determine the memory address allocated by **malloc** during a strdup call, we can utilize **ltrace**, which traces *library function calls*:



This shows strdup places its duplicated string at address 0x0804a008
We'll craft our payload with a shellcode exploit ([this one is only 21 bytes long](#)), followed by padding to reach the return address, and then append 0x0804a008 in little endian.
We just need to determine the right padding, and for this, we'll employ a unique pattern from [this website](#).

