## Problem 3 - (*25 pts*) No Overflow?

Consider the following CTF challenge:

```c
1  void challenge(void) {
2    int8_t size;
3    char flag[16];
4    char buf[128];
5
6    // implementation not shown
7    load_flag(flag);
8
9    memset(buf, 0, 128);
10
11   fread(&size, 1, 1, stdin);
12
13   if (size >= 128) {
14     return;
15   }
16
17   fread(buf, size, 1, stdin);
18   printf("%s", buf);
19 }
```

The `load_flag` function loads the flag data from an access-controlled file on the system.

Answer the following questions:

1. **(5pts)** Complete the following stack diagram assuming the program is paused at line 8.

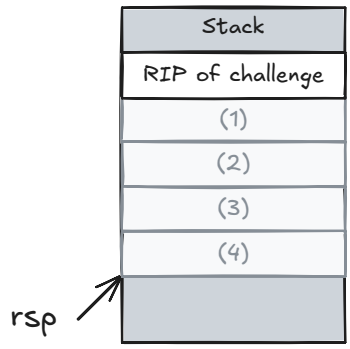


Figure 3: Stack state at line 8

2. **(5pts)** Provide the byte string inputs to `fread` on line 11 and `fread` on line 17 that will give you the flag.

3. **(5pts)** Explain how your solution to the previous part works.
4. **(5pts)** Provide the byte string inputs to `fread` on line 11 and `fread` on line 17 that would allow you to execute the largest possible shellcode payload. Assume all memory safety features are disabled and the compiler allocates 8 bytes of space for `size`.
   - *Use a variable named `SHELLCODE` that contains the shellcode payload as a byte string.*
   - *Use a variable named `BUF_ADDRESS` that contains the address of `buf` as a byte string (presumably reverse-engineered with `gdb`).*
   - *You must specify the length of `SHELLCODE`.*
5. **(5pts)** Explain how your solution to the previous part works.

## Solution:

1. Stack diagram solution:

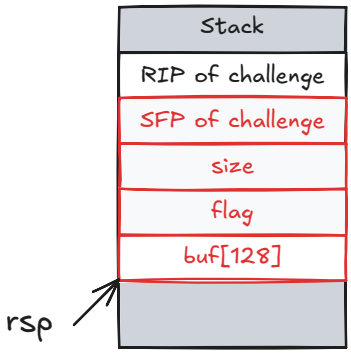


Figure 4: Stack state at line 8

2. Byte string inputs:
   - Line 11 `fread`:

```py
b'\x80' # or any greater unsigned value
```

   - Line 17 `fread`:

```py
b' ' * 128 # or any value greater iff the previous input was exactly b'\x80'
```

3. There is a casting error in the program. Our input is evaluated as signed for the check, but unsigned in `fread`, allowing us to put more than 127 non-null characters in the buffer. If we put exactly 128 non-null characters in the buffer, `printf` will read until the end of `flag` is reached, since it follows immediately after the buffer.
4. Byte string inputs:
   - Line 11 `fread`:

```py
b'\xa8' # 168 or any greater value to get to challenge return address
```

- Line 17 `fread`:

```py
SHELLCODE + BUF_ADDRESS # len(SHELLCODE) == 160 bytes
```

5. If we don't care about overwriting the flag, we can simply write a bunch of shellcode onto the stack, and overwrite the return address of `challenge` with the address of `buf`.