Problem 2 - (*25 pts*) Bypassing Memory Safety Features

Consider the following code snippet:

```c
1  void foo(void) {
2    char foobuf[256];
3    fgets(foobuf, 257, stdin);
4  }
5
6  void bar(void) {
7    foo();
8  }
9
10 void main(void) {
11   char * buf = (char *) malloc(128);
12   fgets(buf, 128, stdin);
13   printf("%p", buf);
14   bar();
15 }
```

Assume the following:

- **ASLR is enabled**.
- All other memory safety features disabled.
- Assume ASLR will always randomize the value of the saved base pointer of `foo` to have a LSB in the inclusive range `0x20 - 0xf8`.
- You can represent the output of `printf` on line 13 with a variable `ADDRESS`. This is an 8-byte, little-endian byte string.
- You can represent shellcode with a variable `SHELLCODE`, which has a maximum length of 120 bytes.
- *Note: SFP = saved frame pointer = saved base pointer (rbp)*

Answer the following questions:

1. **(6pts)** Complete the following stack diagram assuming the program is paused at line 3 (before `fgets` is called).

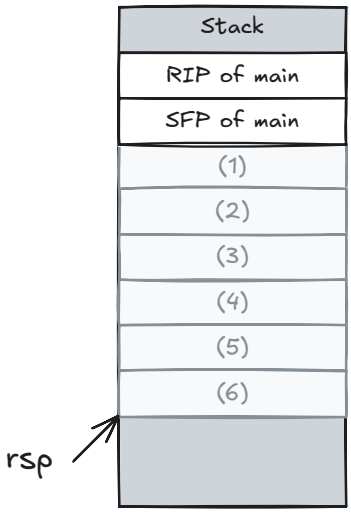

Figure 1: Stack state at line 3

2. **(8pts)** Provide the byte string inputs to `fgets` on line 12 and `fgets` on line 3 that will lead to shellcode execution. You can craft these inputs using Python syntax. For example:

```py
>>> b'\x31' * 8 + b'\x0b\xad\xc0\xde'
b'11111111\x0b\xad\xc0\xde'
>>> ADDR = b'\x00\x00\x7f\xff\xff\xff\x80\x80'
>>> ADDR + b'\xc0\xde'
b'\x00\x00\x7f\xff\xff\xff\x80\x80\x00\xc0\xde'
```

3. **(11pts)** Explain why this exploit is only guaranteed to work if the saved base pointer of `foo` has a LSB in the range `0x20 - 0xf8`.

Solution:

1. The stack diagram should look as follows:

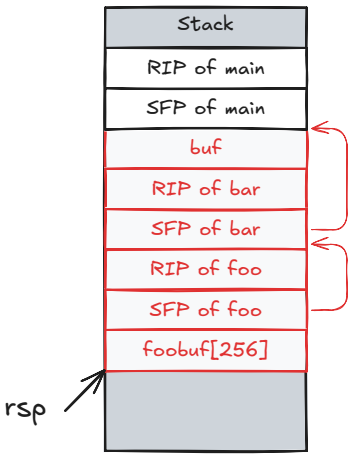

Figure 2: Stack state at line 3

2. The inputs that lead to shellcode execution are as follows:
   - For line 12 `fgets`:

     ```py
     SHELLCODE
     ```

   - For line 3 `fgets`:

     ```py
     ADDRESS * 32
     ```

     ADDRESS is 8 bytes. We're repeating it 32 times => filling up 256 bytes with the same address.

3. How this exploit works: we overwrite the LSB of `foo`'s SFP with the null terminator. Because the LSB is in the range `0x20 - 0xf8`, this points the SFP into 'foobuf' (where we have put our repeated ADDRESS). This exploit is only guaranteed to work when `foo`'s SFP has a LSB in the range `0x20 - 0xf8` because this is the only range for which overwriting the LSB with 0 is guaranteed to give us control over the effective return address of `bar` using `foobuf` (when the function unwind happens, it will point to ADDRESS -> the place in the heap where we put SHELLCODE). There are 4 possible values less than `0x20` that `foo`'s SFP could take that would lead to different outcomes:
   - overwriting `0x00` with 0 would do nothing
   - overwriting `0x08` with 0 would point the modified SFP to `foo`'s saved RIP, which leads to uncontrollable behavior.
   - overwriting `0x10` with 0 would point the modified SFP to itself— also uncontrollable behavior.
   - overwriting `0x18` puts us in `foobuf`, but this still doesn't give us control of the effective return address of `bar`, only `bar`'s SFP.

   Still, even with ASLR, the odds of exploit are pretty high.