Q1 ***Indirection*** **(0 points)**

Consider the follow

```
1  #include <std
2  #include <str
3
4  struct log_en
5      char title[8];
6      char *msg;
7  };
8
9  void log_event(char *title, char *msg) {
10     size_t len = strnlen(msg, 256);
11     if (len == 256) return; /* Message too long. */
12     struct log_entry *entry = malloc(sizeof(struct log_entry));
13     entry->msg = malloc(256);
14     strcpy(entry->title, title);
15     strncpy(entry->msg, msg, len + 1);
16     add_to_log(entry); /* Implementation not shown. */
17 }
```


Assume you are on a little-endian 32-bit x86 system and no memory safety defenses are enabled.

Q1.1 (3 points) Which of the following lines contains a memory safety vulnerability?

- ◯ (A) Line 10
- ◯ (B) Line 13
- ◯ (C) Line 14
- ◯ (D) Line 15
- ◯ (E) ——
- ◯ (F) ——

Q1.2 (3 points) Fill in the numbered blanks on the following stack and heap diagram for `log_event`. Assume that lower-numbered addresses start at the bottom of both diagrams.

Stack

msg

Heap

3
2

- ◯ (G) 1 = `ent` `try->title` 3 = `msg`
- ◯ (H) 1 = `entry->title` 2 = `msg` 3 = `entry->title`
- ◯ (I) 1 = `title` 2 = `entry->title` 3 = `entry->msg`
- ◯ (J) 1 = `title` 2 = `entry->msg` 3 = `entry->title`
- ◯ (K) ——
- ◯ (L) ——

Using GDB, you find that the address of the rip of `log_event` is `0xbfffe0f0`.

Let `SHELLCODE` be a 40-byte shellcode. Construct an input that would cause this program to execute shellcode. Write all your answers in Python 2 syntax (just like Project 1).

Q1.4 (6 points) Give the input for the `title` argument.

Q1.5 (6 points) Give the input for the `msg` argument.

Q2 *Stack Exchange* (19 points)

Consider the following vulnerable C code:

```
1  #include <byteswap.h>
2  #include <inttypes.h>
3  #include <std
4
5  void prepare_
6      char buff
7      int64_t *
8
9      printf("W          er?\n");
10     fread(buffer, 1, 68, stdin);
11
12     printf("What is the pointer?\n");
13     fread(&ptr, 1, sizeof(uint64_t), stdin);
14
15     if (ptr < buffer || ptr >= buffer + 68) {
16         printf("Pointer is outside buffer!");
17         return;
18     }
19
20     /* Reverse 8 bytes of memory at the address ptr */
21     *ptr = bswap_64(*ptr);
22 }
23
24 int main(void) {
25     prepare_input();
26     return 0;
27 }
```


The `bswap_64` function takes in 8 bytes and returns the 8 bytes in reverse order.

Assume that the code is run on a 32-bit system, no memory safety defenses are enabled, and there are no exception handlers, saved registers, or compiler padding.

Q2.1 (3 points) Fill in the numbered blanks on the following stack diagram for `prepare_input`.

1	(0xbffff494)
2	(0xbffff490)
3	(0xbffff450)
4	(0xbffff44c)

- ◯ (A) 1 = `sfp`, 2 = `rip`, 3 = `buffer`, 4 = `ptr`
- ◯ (B) 1 = `sfp`, 2 = `rip`, 3 = `ptr`, 4 = `buffer`
- ◯ (C) 1 = `rip`, 2 = `sfp`, 3 = `buffer`, 4 = `ptr`
- ◯ (D) 1 = `rip`, 2 = `sfp`, 3 = `ptr`, 4 = `buffer`
- ◯ (E) ——
- ◯ (F) ——

Q2.2 (4 points) Which of these values on the stack can the attacker write to at lines 10 and 13? Select all that apply.

☐ (G) `buffer`

☐ (H) `ptr`

☐ (I) `sfp`

☐ (J) `rip`

☐ (K) None of the above

☐ (L) ——

Q2.3 (3 points) Give [illegible] use this program to execute shellcode. At line 10, first input these bytes:

◯ (A) 64-byte shellcode

◯ (B) `\xbf\xff\xf4\x4c`

◯ (C) `\x4c\xf4\xff\xbf`

◯ (D) `\xbf\xff\xf4\x50`

◯ (E) `\x50\xf4\xff\xbf`

◯ (F) ——

Q2.4 (3 points) Then input these bytes:

◯ (G) 64-byte shellcode

◯ (H) `\xbf\xff\xf4\x4c`

◯ (I) `\x4c\xf4\xff\xbf`

◯ (J) `\xbf\xff\xf4\x50`

◯ (K) `\x50\xf4\xff\xbf`

◯ (L) ——

Q2.5 (3 points) At line 13, input these bytes:

◯ (A) `\xbf\xff\xf4\x50`

◯ (B) `\x50\xf4\xff\xbf`

◯ (C) `\xbf\xff\xf4\x90`

◯ (D) `\x90\xf4\xff\xbf`

◯ (E) `\xbf\xff\xf4\x94`

◯ (F) `\x94\xf4\xff\xbf`

Q2.6 (3 points) Suppose you replace `68` with `64` at line 10 and line 15. Is this modified code memory-safe?

◯ (G) Yes ◯ (H) No ◯ (I) —— ◯ (J) —— ◯ (K) —— ◯ (L) ——

This is the end of Q2. Leave the remaining subparts of Q2 blank on Gradescope, if there are any. You have reached the end of the exam.