Q1 *Indirection* (0 points)

Consider the following vulnerable C code:

```
1  #include <stdlib.h>
2  #include <string.h>
3
4  struct log_entry {
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) Seeing an opportunity to exploit this program, you fire up GDB and step into the `log_event` function. Give a GDB command that will show you the address of the rip of the `log_event` function. (Abbreviations are fine.)

Q1.3 (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`
1
rip
sfp
`len`
`entry`

Heap

3
2

◯ (A) 1 = `entry->title` 2 = `entry->title` 3 = `msg`

◯ (B) 1 = `entry->title` 2 = `msg` 3 = `entry->title`

◯ (C) 1 = `title` 2 = `entry->title` 3 = `entry->msg`

◯ (D) 1 = `title` 2 = `entry->msg` 3 = `entry->title`

◯ (E) ——

◯ (F) ——

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.

◯ (A) —— ◯ (B) —— ◯ (C) —— ◯ (D) —— ◯ (E) —— ◯ (F) ——

Q1.6 (3 points) Which of the following defenses on their own would prevent your exploit?

Note: If stack canaries are enabled, you can assume is still the correct address of the RIP.

- ☐ (G) Stack canaries
- ☐ (H) W^X
- ☐ (I) ASLR
- ☐ (J) None of the above
- ☐ (K) ——
- ☐ (L) ——

Q2 *Memory Safety Vulnerabilities* **(23 points)**

Consider the following vulnerable C code:

```
1  #include <stdio.h>
2  #include <string.h>
3
4  struct packet {
5      char payload[300];
6      char format[300];
7  };
8
9  void deploy(struct packet *ptr) {
10     printf(ptr->format, ptr->payload);
11 }
12
13 int main(void) {
14     struct packet p;
15     do {
16         strcpy(p.format, "%s\n");
17         gets(p.payload);
18         deploy(&p);
19     } while (strcmp(p.payload, "END") != 0);
20     // Assume loop always exits for subpart 3.
21     return 0;
22 }
```

Assume you are on a little-endian 32-bit x86 system. Assume that there is no compiler padding or additional saved registers in all subparts. For the first 3 subparts, assume that **no memory safety defenses** are enabled.

Fill in the following stack diagram, assuming that execution has entered the call to `printf`:

RIP of `main`
SFP of `main`
(1a)
(1b)
(1c)
(2a)
(2b)
(2c)
(2d)
RIP of `printf`
SFP of `printf`

Q2.1 (3 points) For (1a), (1b), and (1c):

◯ (A) (1a) - `p.format`; (1b) - `p.payload`; (1c) - `ptr`

◯ (B) (1a) - `p.payload`; (1b) - `p.format`; (1c) - `ptr`

◯ (C) (1a) - `ptr`; (1b) - `p.payload`; (1c) - `p.format`

◯ (D) (1a) - `ptr`; (1b) - `p.format`; (1c) - `p.payload`

◯ (E) ——

◯ (F) ——

Q2.2 (3 points) For (2a), (2b), (2c), and (2d):

◯ (G) (2a) - RIP of `deploy`; (2b) - SFP of `deploy`; (2c) - `&ptr->format`; (2d) - `&ptr->payload`

◯ (H) (2a) - SFP of `deploy`; (2b) - RIP of `deploy`; (2c) - `&ptr->format`; (2d) - `&ptr->payload`

◯ (I) (2a) - `&ptr->payload`; (2b) - `&ptr->format`; (2c) - RIP of `deploy`; (2d) - SFP of `deploy`

◯ (J) (2a) - `&ptr->payload`; (2b) - `&ptr->format`; (2c) - SFP of `deploy`; (2d) - RIP of `deploy`

◯ (K) (2a) - RIP of `deploy`; (2b) - SFP of `deploy`; (2c) - `&ptr->payload`; (2d) - `&ptr->format`

◯ (L) ——

Q2.3 (3 points) For this subpart only, assume that you may only execute one iteration of the while loop and that the call to `printf` will not segfault. For this subpart, assume that no memory safety defenses are enabled.

If the address of `p` is `0x7ff3ec10`, construct an input at line 18 that would cause the program to execute malicious shellcode. You may reference `SHELLCODE` as a 30-byte malicious shellcode. Write your answer in Python 2 syntax (just like in Project 1).

Clarification during exam: Instead of "Line 18," the question should say "Line 17."

For the remaining subparts, assume that **stack canaries are enabled**. Note that this changes the stack diagram!

Q2.4 (5 points) For your exploit, construct a one-line Python helper function `write_byte(addr, byte)` that returns an input for line 17 of the vulnerable C code. This input should ensure that `byte` is written to the address at `addr`. This function may change bytes **above** `addr` (but not below), as long as the correct byte is written at `addr` itself. **The returned input only needs to work for values of `byte` greater than 8.**

Assume that `addr` is given as a 4-byte Python string containing the bytes of the address in little-endian, and assume that `byte` is given as a Python integer between 9 and 255. For example, `write_byte('\xef\xbe\xad\xde', 128)` would be a valid call to this function. Write your answer in Python 2 syntax (just like in Project 1).

```
1 def write_byte(addr, byte):
2     return # Your answer here
```

Hint: You may find the `%c` format specifier useful: Read 4 bytes off the stack and print as a single character.

Q2.5 (5 points) If the address of `p` is `0x7ff3ec10` and the address of the RIP of `main` is `0x7ff3ee68`, construct a series of inputs for repeated calls at line 18 that would cause the program to execute malicious shellcode. Assume that `write_byte` is implemented correctly, and you may call `write_byte` for as many inputs as you would like. Write your answer as a series of `print` statements, all in Python 2 syntax (just like in Project 1).

Hint: You may write hex literals to represent integers in Python, such as `0x36`.

Clarification during exam: Instead of "Line 18," the question should say "Line 17."

Q2.6 (4 points) Which of the following changes, if made on their own, would prevent the attacker from executing malicious shellcode (not necessarily using your exploit above)?

☐ (G) Enabling non-executable pages in addition to stack canaries

☐ (H) Enabling ASLR in addition to stack canaries

☐ (I) Rewriting the code in a memory-safe language

☐ (J) Using `fgets(p.payload, 300, stdin)` instead of `gets(p.payload)` on line 17

☐ (K) None of the above

☐ (L) ——

Q3 *Memory Safety Mitigations* **(12 points)**

Suppose we are on a 64-bit system, and we have an address space of 2^{50} bytes.

Q3.1 (3 points) How many unused bits are available for pointer authentication in each address?

◯ (A) None ◯ (B) 4 ◯ (C) 11 ◯ (D) 14 ◯ (E) 17 ◯ (F) 32

Q3.2 (3 points) Regardless of your answer to the previous part, for the rest of the question, assume that 10 bits are used for pointer authentication in each address.

Additionally, for the rest of the question, assume that 64-bit stack canaries are enabled. The first byte of the stack canary is always a null byte.

Assume the attacker does not have the ability to create their own pointer authentication codes (PACs). How many bits does the attacker have to guess correctly to guess the stack canary and the PAC?

◯ (G) 0 ◯ (H) 10 ◯ (I) 56 ◯ (J) 64 ◯ (K) 66 ◯ (L) 74

Q3.3 (3 points) Now assume that the attacker has a format string vulnerability that lets them read any part of memory while the program is running.

Assume the attacker does not have the ability to create their own PACs. How many bits does the attacker have to guess correctly to guess the stack canary and the PAC?

◯ (A) 0 ◯ (B) 10 ◯ (C) 56 ◯ (D) 64 ◯ (E) 66 ◯ (F) 74

Q3.4 (3 points) Assume the attacker is interacting with a remote system. Provide one defense that would make brute-force attacks infeasible for the attacker. (Please answer in 10 words or fewer.)