

Q1 Indirection**(18 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 = strlen(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?

☐ Line 10

☒ Line 14

☐ Line 13

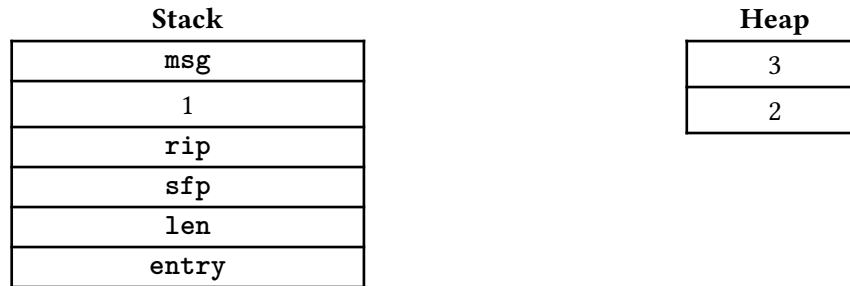
☐ Line 15

Solution: Line 14 uses a `strcpy`, which is not a memory-safe function because it terminates only when it sees a NULL byte, which is under the control of the attacker. Note that line 15 uses a `strncpy` whose length parameter comes from `strlen`, so it is safe.



(Question 1 continued...)

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.



- ☐ 1 = `entry->title` 2 = `entry->title` 3 = `msg`
- ☐ 1 = `entry->title` 2 = `msg` 3 = `entry->title`
- ☒ 1 = `title` 2 = `entry->title` 3 = `entry->msg`
- ☐ 1 = `title` 2 = `entry->msg` 3 = `entry->title`

Solution: The two arguments, `title` and `msg`, must be on the stack, so 1 = `msg`. Structs are filled from lower addresses to higher addresses, so 2 = `entry->title` and 3 = `entry->msg`.

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 of your answers in Python 3 syntax (just like Project 1).

Q1.3 (6 points) Give the input for the `title` argument.

`'A' * 8 + '\xf0\xe0\xff\xbf'`

Solution: The `title` will be used to overflow the `title` buffer in the struct to point the `msg` pointer to the RIP.

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

`'\xf4\xe0\xff\xbf' + SHELLCODE`

Solution: The first 4 bytes will be written in the location of the RIP, which should point to the shellcode.

Q2 Stack Exchange

(19 points)

Consider the following vulnerable C code:

```
1 #include <byteswap.h>
2 #include <inttypes.h>
3 #include <stdio.h>
4
5 void prepare_input(void) {
6     char buffer[64];
7     int64_t *ptr;
8
9     printf("What is the buffer?\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` function¹ 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.

¹Technically, this is a macro, not a function.

(Question 2 continued...)

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)

☐ 1 = `sfp`, 2 = `rip`, 3 = `buffer`, 4 = `ptr`

☒ 1 = `rip`, 2 = `sfp`, 3 = `buffer`, 4 = `ptr`

☐ 1 = `sfp`, 2 = `rip`, 3 = `ptr`, 4 = `buffer`

☐ 1 = `rip`, 2 = `stp`, 3 = `ptr`, 4 = `buffer`

Solution: The `rip` is pushed onto the stack first, followed by the `sfp`, followed by the first local variable `buffer`, followed by the second local variable `ptr`.

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.

☒ `buffer`

☐ `rip`

☒ `ptr`

☐ None of the above

☒ `sfp`

Solution: At line 10, the attacker can write 68 bytes starting at `buffer`. This overwrites all 64 bytes of `buffer` and the 4 bytes directly above it, which is the `sfp`.

At line 13, the attacker can write exactly 1 `uint64_t` * into `ptr`. This overwrites `ptr`, and nothing else.

Notice that the `rip` cannot be directly overwritten

Q2.3 (3 points) Give an input that would cause this program to execute shellcode. At line 10, first input these bytes:

☒ 64-byte shellcode

☐ `\xbf\xff\xf4\x50`

☐ `\xbf\xff\xf4\x4c`

☐ `\x50\xf4\xff\xbf`

☐ `\x4c\xf4\xff\xbf`

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

☐ 64-byte shellcode

☒ `\xbf\xff\xf4\x50`

☐ `\xbf\xff\xf4\x4c`

☐ `\x50\xf4\xff\xbf`

☐ `\x4c\xf4\xff\xbf`

(Question 2 continued...)

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

- | | |
|--|--|
| <input type="radio"/> \xbf\xff\xf4\x50 | <input type="radio"/> \x90\xf4\xff\xbf |
| <input type="radio"/> \x50\xf4\xff\xbf | <input type="radio"/> \xbf\xff\xf4\x94 |
| <input type="radio"/> \xbf\xff\xf4\x90 | <input type="radio"/> \x94\xf4\xff\xbf |

Solution: Line 10 writes 68 bytes into the 64-byte buffer, which lets us overwrite the `sfp`, but not the `rip`.

Line 13 lets us write an arbitrary value into `ptr`, which is then dereferenced in a call to `bswap_64`. This lets us reverse any 8 bytes in memory that we want.

The overarching idea here is to write the address of shellcode in the `sfp`, and then use the call to `bswap_64` to swap the `sfp` and the `rip`.

First, we write the 64 bytes of shellcode into the buffer. Then, we overwrite the `sfp` with `\xbf\xff\xf4\x50`. These bytes are written backwards because `bswap_64` will reverse all 8 bytes of the `sfp` and the `rip`. Finally, we write the address of the `sfp`, `\x90\xf4\xff\xbf` into `ptr`. These bytes are written normally because `bswap_64` never affects `ptr`.

Suppose the current `rip` is `0xdeadbeef`. Our input causes the 8 bytes starting at the `sfp` to be `\xbf\xff\xf4\x50\xef\xbe\xad\xde`. When we call `bswap_64` at the location of `sfp`, the 8 bytes starting at the `sfp` are reversed, so they are now `\xde\xad\xbe\xef\x50\xf4\xff\xbf`. Notice that the `rip` is now pointing to the address of shellcode in the correct little-endian order.

Note: Because you can overwrite the `sfp`, you might be tempted to use the off-by-one exploit from Q4 of Project 1. However, this does not work here because you need enough space to write the shellcode and the address of shellcode in the buffer, but the buffer only has space for the shellcode.

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

- ☐ Yes ☒ No

Solution: No. If you make `ptr` point at one of the last 4 bytes of buffer (which passes the check at line 15), it will cause part of the `sfp` to be overwritten. For example, if `ptr` is located 4 bytes before the end of buffer, the last 4 bytes of buffer will be swapped into the `sfp`.

Because you can overwrite the `sfp`, you could still exploit this modified code using the technique from Project 1, Question 4 (although as mentioned above, you would need shorter shellcode).

Q3 Palindromify

(9 points)

Consider the following C code:

```
1 struct flags {
2     char debug[4];
3     char done[4];
4 };
5
6 void palindromify(char *input, struct flags *f) {
7     size_t i = 0;
8     size_t j = strlen(input);
9
10    while (j > i) {
11        if (input[i] != input[j]) {
12            input[j] = input[i];
13            if (strcmp("BBBB", f->debug, 4) == 0) {
14                printf("Next: %s\n", input);
15            }
16        }
17        i++; j--;
18    }
19 }
20
21 int main(void) {
22     struct flags f;
23     char buffer[8];
24     while (strcmp("XXXX", f.done, 4) != 0) {
25         gets(buffer);
26         palindromify(buffer, &f);
27     }
28     return 0;
29 }
```

Assume you are on a little-endian 32-bit x86 system. Assume that there is no compiler padding or saved registers in all questions.

Here is the function definition for `strcmp`:

```
int strcmp(const char *s1, const char *s2, size_t n);
```

The `strcmp()` function compares the first (at most) `n` bytes of two strings `s1` and `s2`. It returns an integer less than, equal to, or greater than zero if `s1` is found, respectively, to be less than, to match, or be greater than `s2`

(Question 3 continued...)

Q3.1 (3 points) Which of the following lines contains a memory safety vulnerability?

☐ Line 10

☐ Line 24

☐ Line 12

☒ Line 25

Solution: Line 25 contains a vulnerable call to `gets`, which will allow us to overflow `buffer`.

(Question 3 continued...)

Q3.2 (3 points) Which of these inputs would cause the program to execute shellcode located at 0xbffff34d0?

- ☒ '\x00' + (11 * A) + (4 * 'X') + (4 * 'A') + '\xd0\x34\xff\xbf'
- ☐ '\x00' + (19 * 'A') + '\xd0\x34\xff\xbf'
- ☐ (20 * 'X') + '\xd0\x34\xff\xbf'
- ☐ '\x00' + (7 * 'A') + (4 * 'X') + (4 * 'A') + '\xd0\x34\xff\xbf'
- ☐ (16 * 'X') + '\xd0\x34\xff\xbf'
- ☐ None of the above

Solution: First, notice that **buffer** resides in **main**, so we're going to attempt to overwrite the RIP of **main** in this attack. Here's what the stack diagram looks like:

[4] MAIN RIP
[4] MAIN SFP
[4] f.done
[4] f.debug
[8] buffer
...

At a high level, we're going to follow our traditional attack: write past the end of **buffer** and replace the RIP with the address of our shellcode. However, in order to force this program to actually execute that shellcode, there are two **while** loops that we need to break out of.

After our input is copied into **buffer**, we will enter the **palindromify** method. At this point, we need a way to skip the **while** loop that attempts to copy non-matching values from the end of **input** to the beginning - if we don't skip this function, the RIP in our attack will be overwritten by the garbage at the beginning.

To skip this loop, we add a null terminator at the beginning of our exploit - consequently, when **strlen(input)** is called, it will return 0. At this point **j > i** will evaluate to **false**, and we'll skip over the loop.

Then, when the method returns, we need a way to break out of the **while** loop in **main** - otherwise, our program will continue to run forever. To do this, we need to set the **f.done** flag on the stack to **XXXX**.

Because the struct resides above the buffer on stack, we can do this by placing **XXXX** precisely at the location of **f.done**, which resides 12 bytes above **buffer**.

With this information, our exploit looks like:

'\x00' + (11 * 'A') + (4 * 'X') + (4 * 'A') + '\xd0\x34\xff\xbf'

(Question 3 continued...)

Q3.3 (3 points) Assume you did the previous part correctly. At what point will the instruction pointer jump to the shellcode?

- ☐ Immediately after `palindromify` returns
- ☐ Immediately after `gets` returns
- ☒ Immediately after `main` returns
- ☐ Immediately after `printf` returns

Solution: Because we overwrite the RIP in `main`, the shellcode will begin executing when `main` returns.