CS161 Summer 2025

Introduction to Computer Security

Exam Prep 2

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 = 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?

○ Line 10 ○ Line 14

○ Line 13 ○ Line 15

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`
1
`rip`
`sfp`
`len`
`entry`

Heap

3
2

○ 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`

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.

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

Q2 *Stack Exchange* (19 points)

Consider the following vulnerable C code:

```c
#include <byteswap.h>
#include <inttypes.h>
#include <stdio.h>

void prepare_input(void) {
    char buffer[64];
    int64_t *ptr;

    printf("What is the buffer?\n");
    fread(buffer, 1, 68, stdin);

    printf("What is the pointer?\n");
    fread(&ptr, 1, sizeof(uint64_t *), stdin);

    if (ptr < buffer || ptr >= buffer + 68) {
        printf("Pointer is outside buffer!");
        return;
    }

    /* Reverse 8 bytes of memory at the address ptr */
    *ptr = bswap_64(*ptr);
}

int main(void) {
    prepare_input();
    return 0;
}
```

The **`bswap_64`** function[1] 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.

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

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`**

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`**

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`

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

○ `\xbf\xff\xf4\x50`

○ `\x90\xf4\xff\xbf`

○ `\x50\xf4\xff\xbf`

○ `\xbf\xff\xf4\x94`

○ `\xbf\xff\xf4\x90`

○ `\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?

○ Yes

○ No

Q3 *Palindromify* **(9 points)**

Consider the following C code:

```c
struct flags {
    char debug[4];
    char done[4];
};

void palindromify(char *input, struct flags *f) {
    size_t i = 0;
    size_t j = strlen(input);

    while (j > i) {
        if (input[i] != input[j]) {
            input[j] = input[i];
            if (strncmp("BBBB", f->debug, 4) == 0) {
                printf("Next: %s\n", input);
            }
        }
        i++; j--;
    }
}

int main(void) {
    struct flags f;
    char buffer[8];
    while (strncmp("XXXX", f.done, 4) != 0) {
        gets(buffer);
        palindromify(buffer, &f);
    }
    return 0;
}
```

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 `strncmp`:

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

```
The strncmp() 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 12

○ Line 24

○ Line 25

Q3.2 (3 points) Which of these inputs would cause the program to execute shellcode located at `0xbfff34d0`?

○ `'\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

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 `main` returns

○ Immediately after `gets` returns

○ Immediately after `printf` returns