Q1 *Echo, Echo, Echo* **(20 points)**

Consider the following vulnerable C code:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  char name[32];
5
6  void echo(void) {
7      char echo_str[16];
8      printf("What do you want me to echo back?\n");
9      gets(echo_str);
10     printf("%s\n", echo_str);
11 }
12
13 int main(void) {
14     printf("What's your name?\n");
15     fread(name, 1, 32, stdin);
16     printf("Hi %s\n", name);
17
18     while (1) {
19         echo();
20     }
21
22     return 0;
23 }
```

The declarations of the used functions are as given below.

```
1  // execute the system command specified in 'command'.
2  int system(const char *command);
```

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

Q1.1 (2 points) Assume that execution has reached line 8. Fill in the following stack diagram. Assume that each row represents 4 bytes.

Stack

1
2
RIP of `echo`
SFP of `echo`
3
4

◯ (A) (1) - RIP of `main`; (2) - SFP of `main`; (3) - `echo_str[0]`; (4) - `echo_str[4]`

◯ (B) (1) - SFP of `main`; (2) - RIP of `main`; (3) - `echo_str[0]`; (4) - `echo_str[4]`

● (C) (1) - RIP of `main`; (2) - SFP of `main`; (3) - `echo_str[12]`; (4) - `echo_str[8]`

◯ (D) ——

◯ (E) ——

◯ (F) ——

Solution: The first two items on the stack are the RIP and SFP of `main`, respectively. Since the stack grows down, lower addresses are at the bottom of the diagram, and arrays are filled from lower addresses to higher addresses and are zero-indexed. As such, row (3) contains `echo_str[12]`, and row (4) contains `echo_str[8]`.

Q1.2 (3 points) Using GDB, you find that the address of the RIP of `echo` is `0x9ff61fc4`.

Construct an input to `gets` that would cause the program to execute malicious shellcode. Write your answer in Python syntax (like in Project 1). You may reference `SHELLCODE` as a 16-byte shellcode.

Solution: Where to put the `SHELLCODE` does not matter. This is a simple stack-smashing attack: we want to redirect execution to `SHELLCODE` when the `echo` function returns.

Approach 1: Place the Shellcode in the Buffer

```
SHELLCODE + 'A' * 4 + '\xb0\x1f\xf6\x9f'
```

Approach 2: Place the Shellcode above the RIP

```
'A' * 20 + '\xc8\x1f\xf6\x9f' + SHELLCODE
```

There may be a few other correct answers here (with the shellcode placed at slightly different offsets within the buffer or above the RIP), but these are the most common.

Q1.3 (4 points) Which of the following defenses on their own would prevent an attacker from executing the exploit above? Select all that apply.

■ (A) Stack canaries

■ (B) Pointer authentication

■ (C) Non-executable pages

■ (D) ASLR

☐ (E) None of the above

☐ (F) ——

Solution: Stack canaries defend against this attack because we are consecutively writing from the local variables to the RIP. The canary would be checked when the `echo` function returns, and because we don't have a way to leak the value of the canary in our exploit, canaries effectively stop our exploit from succeeding.

Non-executable pages defend against our exploit by preventing the shellcode on the stack (a write-not-execute region of memory) from being executed.

If ASLR were enabled, we wouldn't be able to reliably find the address of the RIP - it would change every time! We'd have to use one of our special attacks specifically for ASLR to bypass this (e.g. ROP). As such, ASLR stops our original exploit from succeeding.

Pointer authentication would require us to forge a valid pointer authentication code along with our new RIP. We don't have a way to do this, so pointer authentication stops our exploit from succeeding.

Note: we received a handful of questions asking if the "Pointer Authentication" answer choice was referring to a 32-bit system, which is what was stated in the prologue of this question, or a 64-bit system, which is what we originally intended. As such, we awarded credit for both answer choices.

Q1.4 (5 points) Assume that non-executable pages are enabled so we cannot execute SHELLCODE on stack. We would like to exploit the `system(char *command)` function to start a shell. This function executes the string pointed to by `command` as a shell command. For example, `system("ls")` will list files in the current directory.

Construct an input to `gets` that would cause the program to execute the function call `system("sh")`. Assume that the address of `system` is `0xdeadbeef` and that the address of the RIP of `echo` is `0x9ff61fc4`. Write your answer in Python syntax (like in Project 1).

Hint: Recall that a return-to-libc attack relies on setting up the stack so that, when the program pops off and jumps to the RIP, the stack is set up in a way that looks like the function was called with a particular argument.

Solution: Our goal is to make `echo` return to the system function by changing the `RIP of echo` to the address of `system`. When `echo` returns to `system`, the stack should look like the stack diagram below, because by calling convention the callee expects its arguments and its `RIP` to be pushed onto the stack by the caller. **It's the callee's responsibility to push the SFP onto the stack as its first step.**

Therefore we need to first place garbage bytes from the beginning of `name` up to the `RIP of echo` (`'A' * 20`) and replace the `RIP of echo` with the address of `system` (`'\xef\xbe\xad\xde'`) so that `echo` will return to `system`. Now, we want to create the stack diagram above to make the stack in line with what the `system` method expects. Thus, we add four bytes of garbage where the `system` method expects `RIP of system` to be. Note that, `RIP of system` is the address that `system` method will return to. Then, we place the address of "sh" at the location where `system` expects an argument, and place the string "sh" at that location (which is 8 bytes above the `RIP of system`).

Stack

`command` (pointer to `"sh"`)
(Expected) RIP of `system`

As such, our exploit may look something like the following:

```
'A' * 20 + '\xef\xbe\xad\xde' + 'B' * 4 + '\xd0\x1f\xf6\x9f'
+ 'sh' + '\x00'
```

NOTE: Since the stack below the `RIP of echo` will get invalidated (because it's below the ESP) after `echo` returns, we cannot make any assumptions about whether the values placed there would remain as is. Therefore, you should not place the string "sh" in `name`.

Q1.5 (6 points) Assume that, in addition to non-executable pages, ASLR is also enabled. However, addresses of global variables are not randomized.

Is it still possible to exploit this program and execute malicious shellcode?

◯ (A) Yes, because you can find the address of both `name` and `system`

◯ (B) Yes, because ASLR preserves the relative ordering of items on the stack

◯ (C) No, because non-executable pages means that you can't start a shell

● (D) No, because ASLR will randomize the code section of memory

◯ (E) —

◯ (F) —

Solution: If ASLR is enabled, the address of `system`, a line of code in the *code* section of memory, will be randomized each time the program is run. Because our exploit uses this address, ASLR will effectively prevent us from using our approach!

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

Solution: Stack diagram:

```
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[64:68] + '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.