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

Consider the following vulnerable C code:

```c
#include <stdio.h>
#include <stdlib.h>

char name[32];

void echo(void) {
    char echo_str[16];
    printf("What do you want me to echo back?\n");
    gets(echo_str);
    printf("%s\n", echo_str);
}

int main(void) {
    printf("What's your name?\n");
    fread(name, 1, 32, stdin);
    printf("Hi %s\n", name);

    while (1) {
        echo();
    }

    return 0;
}
```

The declarations of the used functions are as given below.

```c
// execute the system command specified in 'command'.
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)  

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.

```

```

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) —
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.*

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) ——
Consider the following vulnerable C code:

```c
typedef struct {
    char mon[16];
    char chan[16];
} duo;

void third_wheel(char *puppet, FILE *f) {
    duo monlder;
    duo richard;
    fgets(richard.mon, 16, f);
    strcpy(richard.chan, puppet);
    int8_t alias = 0;
    size_t counter = 0;

    while (!richard.mon[15] && richard.mon[0]) {
        size_t index = counter / 10;
        if (monlder.mon[index] == 'A') {
            monlder.mon[index] = 0;
        }
        alias ++;
        counter ++;
        if (counter == ___ || counter == ___) {
            richard.chan[alias] = monlder.mon[alias];
        }
    }

    printf("%s\n", richard.mon);
    fflush(stdout); // no memory safety vulnerabilities on this line
}

void valentine(char *tape[2], FILE *f) {
    int song = 0;
    while (song < 2) {
        read_input(tape[song]); // memory-safe function, see below
        third_wheel(tape[song], f);
        song ++;
    }
}
```
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 0xffffffffd84.
- 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

```

```

This content is protected and may not be shared, uploaded, or distributed.
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.*

[Blank Box]

[Blank Box]

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]:

[Blank Box]

Provide an input to fgets in third_wheel:

[Blank Box]
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]:

Provide an input to fgets in third_wheel: