Problem 7 ***Fail Caesar*** **(12 points)**

A student at a well known Junior University decided to write their own Caesar Cipher after learning about them in their computer security class. Unfortunately for the student, they fell asleep during the lecture on memory safety. (Note: The `atoi()` function converts the initial portion of the string to an integer, returning 0 in case of an error.)

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  void encrypt(int offset, char plaintext[]) {
4      char ciphertext[64];
5      memset(ciphertext, 0, 64);
6      int i = 0;
7      fgets(plaintext, 64, stdin);
8      while (plaintext[i]){
9          ciphertext[i] = plaintext[i] + offset;
10         i++;
11     }
12     printf(ciphertext);
13 }
14
15 int main(int argc, char *argv[]){
16     char buffer[64];
17     int offset = 0;
18     if (argc > 1) offset = atoi(argv[1]) % 26;
19     while (!feof(stdin)){
20         memset(buffer, 0, 64);
21         encrypt(offset, buffer);
22     }
23     return 0;
24 }
```

(a) What line contains a memory vulnerability? What is this vulnerability called?

(b) Give a file that, when input to the command `failcaesar` with no arguments, will cause the program to crash.

(c) How would you change the line to fix the vulnerability?

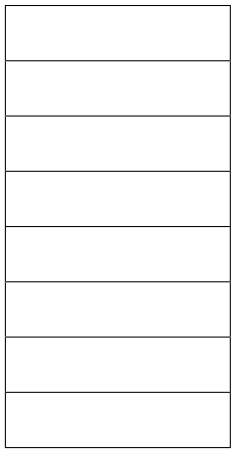
(d) The student's friend who was awake for the memory safety lecture tells them to enable stack canaries to make their code more secure. If an attacker does not have time to perform a bruteforce attack, does enabling stack canaries prevent this code from being exploited? Explain why or why not.

Problem 7 ***Memory safety exploits*** **(26 points)**

The following code allows you to print characters of your choice from a string. It runs on a 32-bit x86 system with **stack canaries enabled**, but no other memory defense methods in use. Assume local variables are pushed onto the stack in the order that they are declared, and there is no extra padding, saved registers, or exception handlers. (These are the same assumptions as in homework 1.) Note that `scanf("%d", &offset)` reads a number from the input, converts it to an integer, and stores it in the `offset` variable.

```
1  void foo() {
2    char buf[300];
3    gets(buf);
4  }
5
6  int main() {
7    char *ptr;
8    int offset = 0;
9    char important[12] = "sEcuRitY!!!";
10   while (offset >= 0) {
11     scanf("%d", &offset);
12     ptr = important + offset;
13     printf("%c\n", *ptr);
14   }
15   foo();
16   return 0;
17 }
```

(a) Draw the stack, when at the point in time when line 12 of the code is executing, by filling in the diagram below. Label the location of sfp, rip (saved return address), stack canary, and the `ptr`, `offset`, and `important` variables, for `main`'s stack frame. Each empty box represents 4 bytes of stack memory. If a value spans multiple boxes, label all of them.



(b) Peyrin informs you that this code contains a vulnerability which leaks the value of `main`'s stack canary. Which sequence of inputs would leak this information? Fill in the blanks below.

_____ \n _____ \n _____ \n _____ \n

(c) Next, suppose you want to develop a reliable arbitrary-code-execution exploit that works by overwriting `foo`'s entire return address, so that when `foo` returns, your shellcode will be executed. You first supply the string from part (b) to learn the value of the stack canary, followed by the string '-1\n', followed by a carefully chosen third string of some length. Write the *minimum* possible length of the third string, to achieve this. Assume your shellcode is 100 bytes long and it cannot be shortened.

(d) Your friend claims that it's not necessary to overwrite the entire return address to achieve arbitrary code execution: if you don't get unlucky with where certain addresses happen to fall, it's possible to reduce the length of the third string in part (c) to 304 bytes or 305 bytes, using an exploit that overwrites the least significant byte of sfp. Is she right?

○ Yes ○ No

(e) The developers propose to fix the program by replacing lines 12–13 with the following code. Fill in the blank inside the if-statement to make the fix correct.

```
12    if (_____________________________________) {
13      ptr = important + offset;
14      printf("%c\n", *ptr);
15    }
```

Unfortunately the fix isn't available yet. Unsettled by your exploit, the sysadmins **enable ASLR** for the stack and the heap as a temporary defense for the rest of this question.

You discover that the code (text segment) is not randomized, and you learn the address of a `ret` instruction. For the purpose of this question, you can assume that `ret` is a one-word instruction which is equivalent to `pop %eip`. In other words, it loads the instruction at `$esp` into the `$eip` and increments `$esp` by one word.

(f) Which exploit technique would be appropriate for an arbitrary code execution exploit against this code, given this new information?

○ ROP ○ Overwrite the first byte of sfp

○ TOCTTOU ○ Exploit a format string vulnerability

(g) Provide bounds on x, such that the input 'x\n' will cause `ptr` to point somewhere in the region where `buf` will appear.

_____ ≤ x ≤ _____

(h) Your exploit constructs an input as follows: first supply the string from part (b) to learn the value of the stack canary, followed by the string 'x\n' (with x chosen somehow based on part (g)) to set `ptr` appropriately, followed by a carefully chosen third string that is composed from multiple pieces. Below, select all possibilities for how to choose the third string so that the shellcode will be executed with probability at least 1/2.

Assume SHELLCODE is a 100-byte string containing the shellcode you want to execute, CANARY is the 4-byte value of the canary (learned using the technique from part (a)), `gadget` is the 4-byte address of the `ret` instruction you found, and NOPSLED is a 200-byte string containing many NOP instructions. Beware that `gets` will replace the newline at the end of your third string with a null byte, so your exploit might need to deal with this.

1. First 300 bytes of the third string:

- ☐ SHELLCODE * 3
- ☐ SHELLCODE + 'a' * 196 + CANARY
- ☐ NOPSLED + SHELLCODE
- ☐ `gadget` * 75

2. Next 12 bytes of the third string:

- ○ `gadget` * 3
- ○ `gadget` * 2 + CANARY
- ○ CANARY * 3
- ○ CANARY + 'a' * 4 + CANARY
- ○ CANARY + `gadget`*2
- ○ CANARY * 2 + 'a' * 4

3. Next bytes of the third string: (fill in the blank with a Python expression; your expression may reference SHELLCODE, NOP, CANARY, `gadget`, and 'a's, though you won't need them all)

4. Final byte of the third string:
\n