Lab 8

Jarred Parr Alexander Fountain

1. This is a buffer overrun issue. Because the proper memory was not properly allocated, the length of the string overflows into other parts of memory that may not have been originally designated to the program. As a result, this causes problems. On a more high-scale project, this could be catastrophic and open the code to injection issues that would potentially cause harm to the target computer or exploit the software as a whole. The error in this file begins with the malloc of size 16. The problem here is that when you put in `notarealusername` the length is too large, and on the `scanf` line you see the data get placed into that memory incorrectly as a result.

Corrected (VERY ROBUST) Code

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

#define SIZE 16

char* infinite_length_alloc(FILE* input_stream, size_t input_size) {
  char* new_string;
  int ch;
  size_t len = 0;
  new_string = malloc(input_size);

  while ((ch = fgetc(input_stream)) != EOF && ch != '\n') {
    new_string[len++] = ch;
    if (len == input_size) {
      // Double the size of the input string
      new_string = realloc(new_string, input_size * 2);
    }
  }

  // Add our nullbyte
  new_string[len++] = '\0';

  return new_string;
}

int main() {
    char *data1;

    data1 = malloc (SIZE);
    printf ("Please input username: ");
    data1 = infinite_length_alloc(stdin, SIZE);
    printf ("you entered: %s\n", data1);
    free (data1);
    return 0;
```

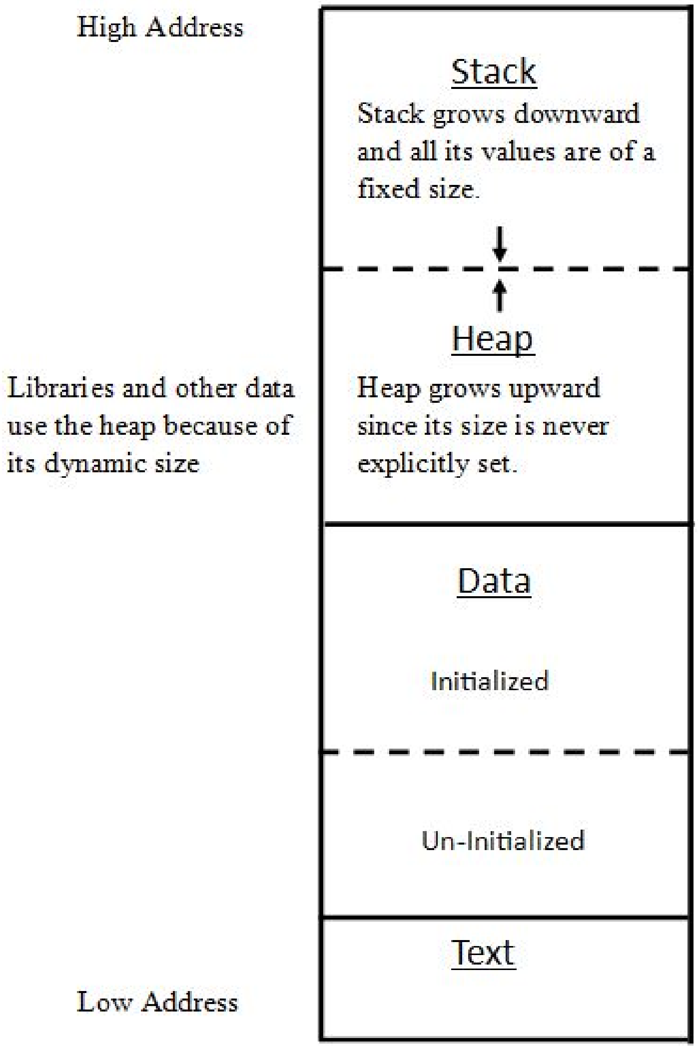
```
}
```

2. Address obfuscation is a tool used by the program (and sometimes the operating system) to obscure the location of critical code. In binary exploit attacks, the attacker can take the executable and reverse engineer its functionality and take advantage of potential memory issues. As a result, this can lead to many problems like code injection vulnerabilities as we saw in problem 1 above. To combat this, address obfuscation is used to allow the program more security by changing the usual pattern of frame mapping to prevent someone from easily tracing the binary output of the program and being able to inject their malicious code.

mem_sample.c

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

// Unitialized variable stored in uninitialized data segment
int global;

// Initialized data in the initiailzed segment
int global2 = 10;

int main(int argc, char** argv) {
  // Uninitialized variable stored in uninit segment
  static int i;
  // Stored in initialized data segment
  static int j = 100;

  // Dynamically made data which resides on the heap
  char* dyno = malloc(16);

  // Statically made data which resides on the stack
  int stati[2] = {1, 2};

  // Memory address of dyno
  char** addr = &dyno;

  // Free the dynamic memory
  free(dyno);

  return 0;
}
```

Stack Drawing



Example Memory Locations

Stack Variable 1	0x7ffe4331c8e0
Stack Variable 2	0x7ffe4331c8d4
Library	0x7fde43048000
Runtime Variable 1	0x55e2ce162260
Runtime Variable 2	0x55e2ce162280
Heap Variable	0x55e2ce162000
Un-initialized Variable 1	0x55e2cca87050
Un-initialized Variable 2	0x55e2cc98704c
Initialized Variable 1	0x55e2cca87040
Initialized Variable 2	0x55e2cca87044
Text	0x55e2cca84000