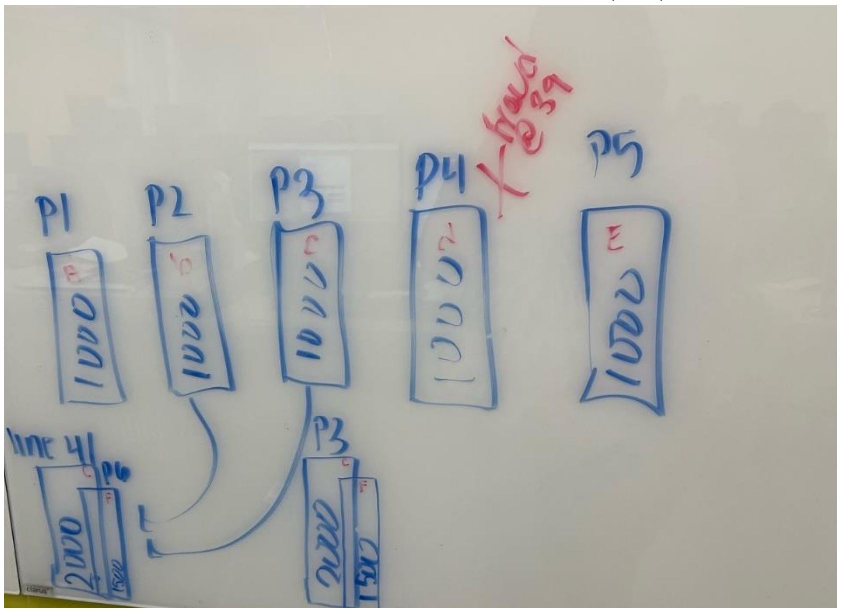
1. Problem statement:
    a. You have been given a program called Exam2.c. The code is complete and will allocate heap space. Through using dynamic allocation and freeing memory is getting overwritten. You do not need to make an exploit. Take the code and add comments explain what the code is doing and figure out why memory is getting overwritten. There is one line in particular where the bug is.
2. Introduction:

In our previous analysis, we discussed stack memory which temporarily stores all of our local variables, methods, and reference variables in memory. is a temporary storage memory. When the computer completes the task, the variable's memory is automatically erased using static memory allocation which follows the last in first out principle (LIFO). The memory of the program also stores heap, data, and text. The heap is a memory used to store global variables. Heap memory utilizes Dynamic memory allocation which is When a running program asks the operating system to give it a block of main memory. This memory is subsequently used by the program. The main difference between stack and heap memory is that stack memory is automatically freed unlike heap memory users must free memory. It becomes significantly more difficult to keep track of whether parts of the heap are allocated or free at any given time as a result.

In order to allocate memory for heap memory in C++ programmer needs to declare the variable using the new operator. Programmer must deallocate the memory using the free() function. If programmer fails to deallocate the memory it will potentially lead to memory leakage. When memory is dynamically allocated and never deallocated, memory leaks happen. The malloc or calloc functions in C programs allocate new memory, and the free function deallocates it. In C++, the new operator is typically used to allocate new memory, whereas the delete or delete [] operator is typically used to deallocate it. The issue with memory leaks is that they build up over time and, if ignored, may cause a program to become unresponsive or even crash.

Executing the incorrect delete operator is among the most frequent mistakes that causes memory leaks. In contrast to the delete [] operator, which should be used to release an array of data values, the delete operator should be used to release a single allocated class or data value. The free function in C does not distinguish between these things. Another typical fault is to overwrite a variable that contains dynamic memory without first freeing any memory that has already been allocated.

A heap overflow is a type of buffer overflow that occurs when data is written to memory that has been allocated for the heap without first doing any bound checking on the data. This may result in the virtual function table being overwritten, as well as any crucial heap data structures such as heap headers or other data that is based on the heap, such as dynamic object references. Heap overflows can occur if memory leaking occurs when memory is continually allocated but is not made available for usage by other programs. Memory leakage might happen if this memory space is not released after use. Heap overflow can also occur if programmer allocates too many variables.

3. Errors in the code
    a. Identify the one line of code where the bug is and explain why the bug is there
        i. Line 41:
            1. **(unsigned int *)((unsigned char *)p1 + real_size_p1) = real_size_p2 + real_size_p3 + prev_in_use + sizeof(size_t) * 2;*
            2. I somewhat understand what the line is doing is allocating the beginning of p2 (which is the end of p1) and storing that for p6. The issue is that p2 is not freed and is a smaller chunk than p6 by 1000 bytes which causes p6 to run into the p3 chunk. This bug here is that at the point in the code only p4 memory has been freed. The bug is that these chunks of memory are overlapped by various variables because the programmer did not free p2 and p3. This line of code causes issues down the line for lines 46, 55, and 58.
            3. 

                a. This is how I visualized what was happening with line 41. P6 starts at the beginning of p2 and ends in the middle of p3.
                b. Note: p3 should really say 1000 bytes and 500 bytes have “f”

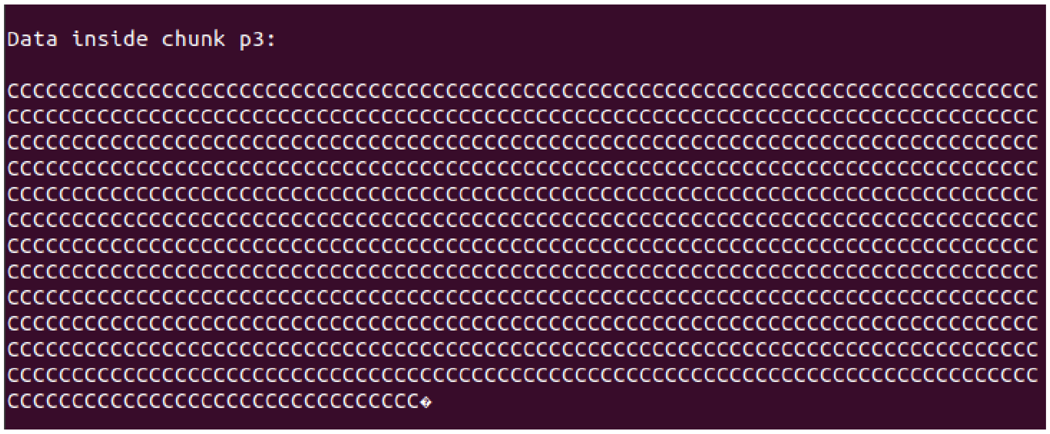
4. Proposed solutions

One solution is if I were to delete line 41 out of code. It does not matter if I freed p2 or p3 it will not use the memory addresses of those since the heap needs to allocate an entire chunk because it is not a stack that follows locating variable right after each other in the memory. After deleting the line of code, the program assigns p6 to a different chunk not inside of p2 or p3 which allows for p3 not to be overwritten in any capacity.

Line 41 wants the program to assigned p6 memory as if it is assigning in static memory allocation (stack memory), but we are dealing with heap memory.

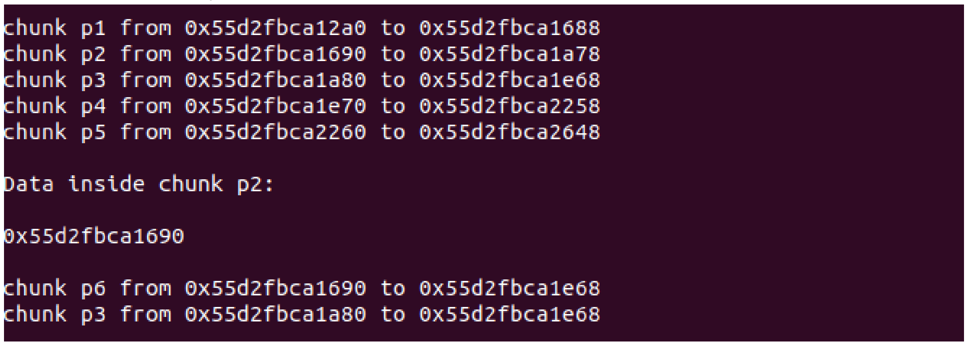
Another solution would be too free p2 and move lines 46-48 before line 41 and change line 46 to be p6 = malloc(1000) and line 55 to be memset(p6, “F”,1000). These changes will allow for p6 chunk to be stored within p2 chunk. I am unsure of the programmer's goal for this program. Depending on the end goal this could work if they simply wanted p6 to be within p2 chunk. If we free p4 or any other chunk p6 will replace those memory addresses like how it did for p2. This is possible because memory in dynamic memory allocation is freed during runtime.

Since p1 –p6 are pointers that point to memory addresses when line 41 was implemented it causes p6 to point to the memory address of both p2 and p3 which is why when you print out p3 it is the same as printing out p6. I believe our OS can’t combine the chunks of memory needed to fulfill line 46, so line 41 will cause it to combine p2 and p3 to allocate for p6. What it needs to do is allocate a brand new chunk of memory.

a. 

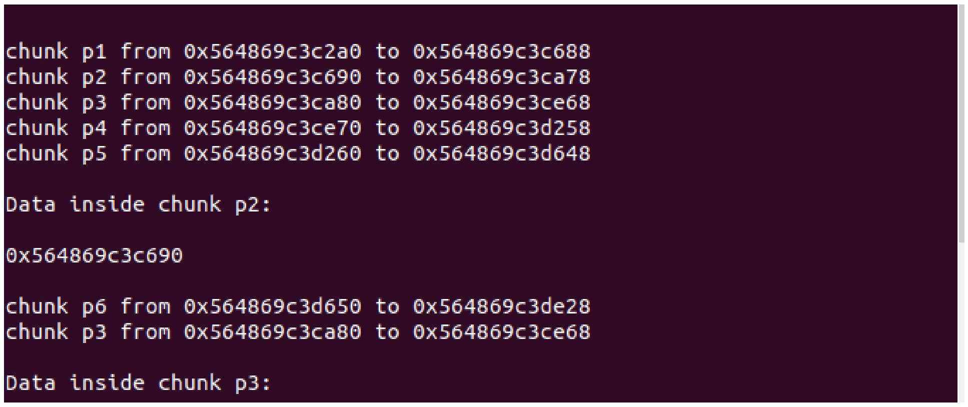
```
Data inside chunk p3:

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC�
```

   i. This was p3 chunk before line 41 consist of all “C"

b. 

```
chunk p1 from 0x55d2fbca12a0 to 0x55d2fbca1688
chunk p2 from 0x55d2fbca1690 to 0x55d2fbca1a78
chunk p3 from 0x55d2fbca1a80 to 0x55d2fbca1e68
chunk p4 from 0x55d2fbca1e70 to 0x55d2fbca2258
chunk p5 from 0x55d2fbca2260 to 0x55d2fbca2648

Data inside chunk p2:

0x55d2fbca1690

chunk p6 from 0x55d2fbca1690 to 0x55d2fbca1e68
chunk p3 from 0x55d2fbca1a80 to 0x55d2fbca1e68
```

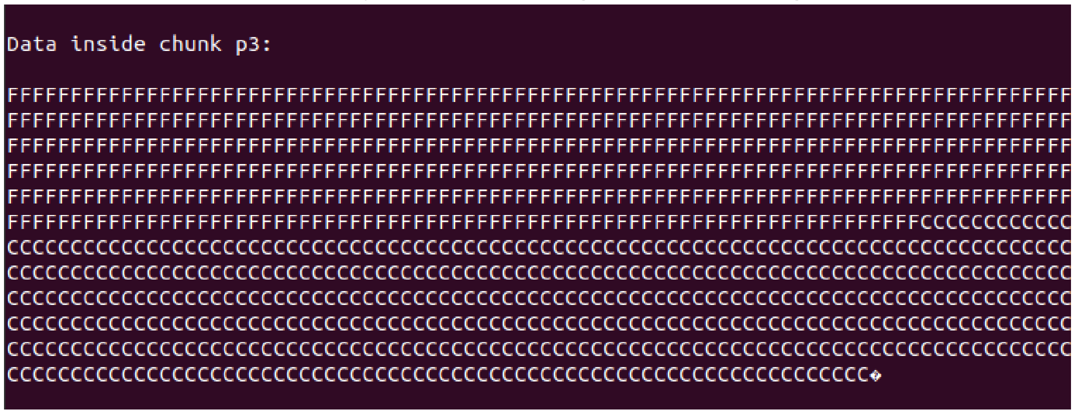
   i. This the chunks with line 41 included
      1. P6 = p2 +p3

c. 

```
chunk p1 from 0x564869c3c2a0 to 0x564869c3c688
chunk p2 from 0x564869c3c690 to 0x564869c3ca78
chunk p3 from 0x564869c3ca80 to 0x564869c3ce68
chunk p4 from 0x564869c3ce70 to 0x564869c3d258
chunk p5 from 0x564869c3d260 to 0x564869c3d648

Data inside chunk p2:

0x564869c3c690

chunk p6 from 0x564869c3d650 to 0x564869c3de28
chunk p3 from 0x564869c3ca80 to 0x564869c3ce68

Data inside chunk p3:
```

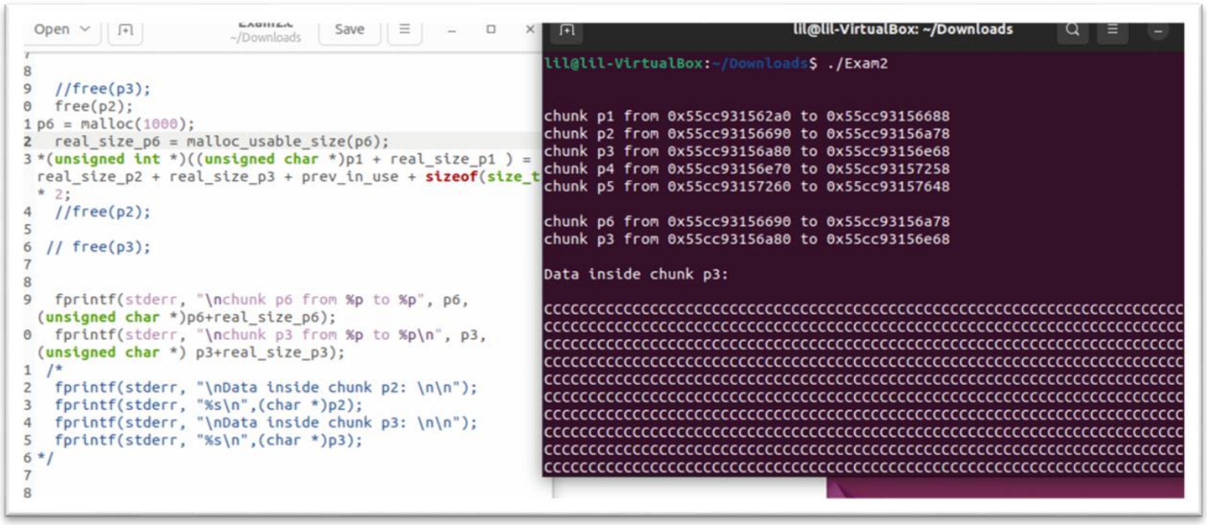
   i. This is the chunks without line 41 included

      1. This is the output with discarding line 41 or moving line 43 above line 41

d. 

```
Data inside chunk p3:

FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC�
```

   i. This was p3 chunk after line 4 we can see that p6 run into p3 chunks and overwrites 500 values of “c” into “f”

e. 

```

  //free(p3);
  free(p2);
p6 = malloc(1000);
  real_size_p6 = malloc_usable_size(p6);
*(unsigned int *)((unsigned char *)p1 + real_size_p1 ) =
real_size_p2 + real_size_p3 + prev_in_use + sizeof(size_t
* 2;
  //free(p2);

 // free(p3);


  fprintf(stderr, "\nchunk p6 from %p to %p", p6,
(unsigned char *)p6+real_size_p6);
  fprintf(stderr, "\nchunk p3 from %p to %p\n", p3,
(unsigned char *) p3+real_size_p3);
 /*
  fprintf(stderr, "\nData inside chunk p2: \n\n");
  fprintf(stderr, "%s\n",(char *)p2);
  fprintf(stderr, "\nData inside chunk p3: \n\n");
  fprintf(stderr, "%s\n",(char *)p3);
*/
```

```
lil@lil-VirtualBox:~/Downloads$ ./Exam2


chunk p1 from 0x55cc931562a0 to 0x55cc93156688
chunk p2 from 0x55cc93156690 to 0x55cc93156a78
chunk p3 from 0x55cc93156a80 to 0x55cc93156e68
chunk p4 from 0x55cc93156e70 to 0x55cc93157258
chunk p5 from 0x55cc93157260 to 0x55cc93157648

chunk p6 from 0x55cc93156690 to 0x55cc93156a78
chunk p3 from 0x55cc93156a80 to 0x55cc93156e68

Data inside chunk p3:

CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
```

   i. Solution to the second solution

5. Conclusion and analysis of solutions

   a. In conclusion, when handling heap memory allocation, it is crucial for programmers to free the allocated memory chunks, or it can cause the program to have vulnerabilities. It is a quick task to do but when forgotten about causes vulnerabilities such as heap overflow which can cause memory leaks. This was demonstrated within this program with line 41 this one line caused the overlapping of chunks of memory to the point that p3 original values of "C" were overwritten. In P3 the first 1000 bytes contained the value "F" and the last 500 bytes contained the value "C" because p6 ran into p3 due to line 41. It is hard to propose a solution to this bug without knowing the purpose of the program.

6. Sources:

a. https://ctf101.org/binary-exploitation/heap-exploitation/
b. https://linux.die.net/man/3/malloc_usable_size
c. https://www.tutorialspoint.com/c_standard_library/c_function_malloc.htm
d. https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/
e. https://www.tutorialspoint.com/what-is-a-malloc-function-in-c-language#:~:text=The%20malloc()%20function%20stands,points%20to%20the%20memory%20location.
f. https://www.educba.com/unsigned-int-in-c/
g. https://resources.infosecinstitute.com/topic/heap-overflow-vulnerability-and-heap-internals-explained/
h. https://www.geeksforgeeks.org/heap-overflow-stack-overflow/
i. https://blog.qualys.com/vulnerabilities-threat-research/2021/01/26/cve-2021-3156-heap-based-buffer-overflow-in-sudo-baron-samedit
j. https://www.design-reuse.com/articles/25090/dynamic-memory-allocation-fragmentation-c.html#:~:text=In%20C%2C%20dynamic%20memory%20is,pointer%20to%20the%20allocated%20memory.