ASSIGNMENT – 2

Special Topic :- Secure Programming with C

SRN: PES1UG19CS542
Name: Trisha Jain
Section: I2
Mail Id: jtrisha0403@gmail.com

Question 1 :

1a) Review the below code:

i) With & without Valgrind tool and make observation with both.
ii) Also provide input with varying size & make your observation.

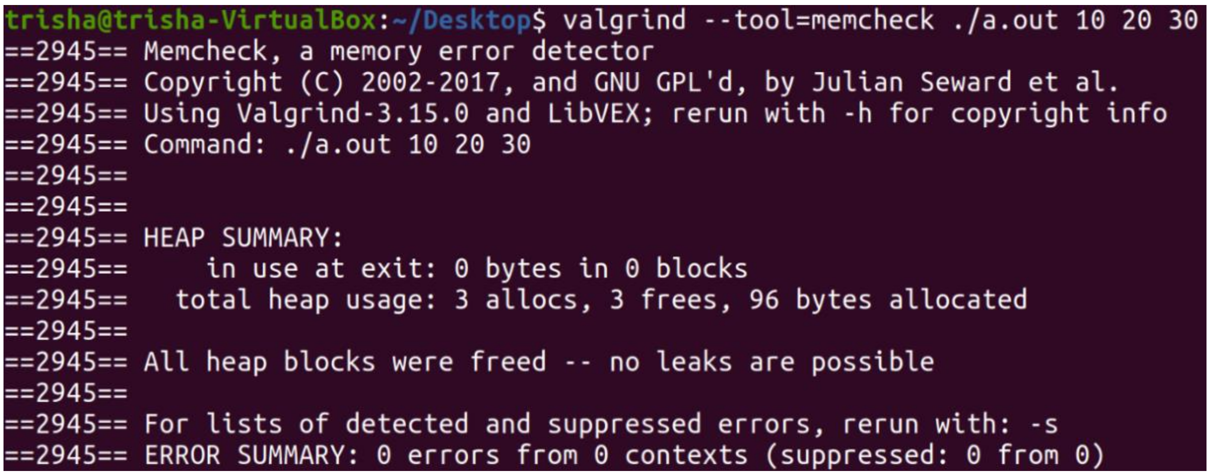
Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char **argv) {
        char *a;
        char *b;
        char *c;
        a = malloc(32);
        b = malloc(32);
        c = malloc(32);
        strcpy(a, argv[1]);
        strcpy(b, argv[2]);
        strcpy(c, argv[3]);
        free(c);
        free(b);
        free(a);
        return 0;
}
```

Solution :-

With Valgrind :

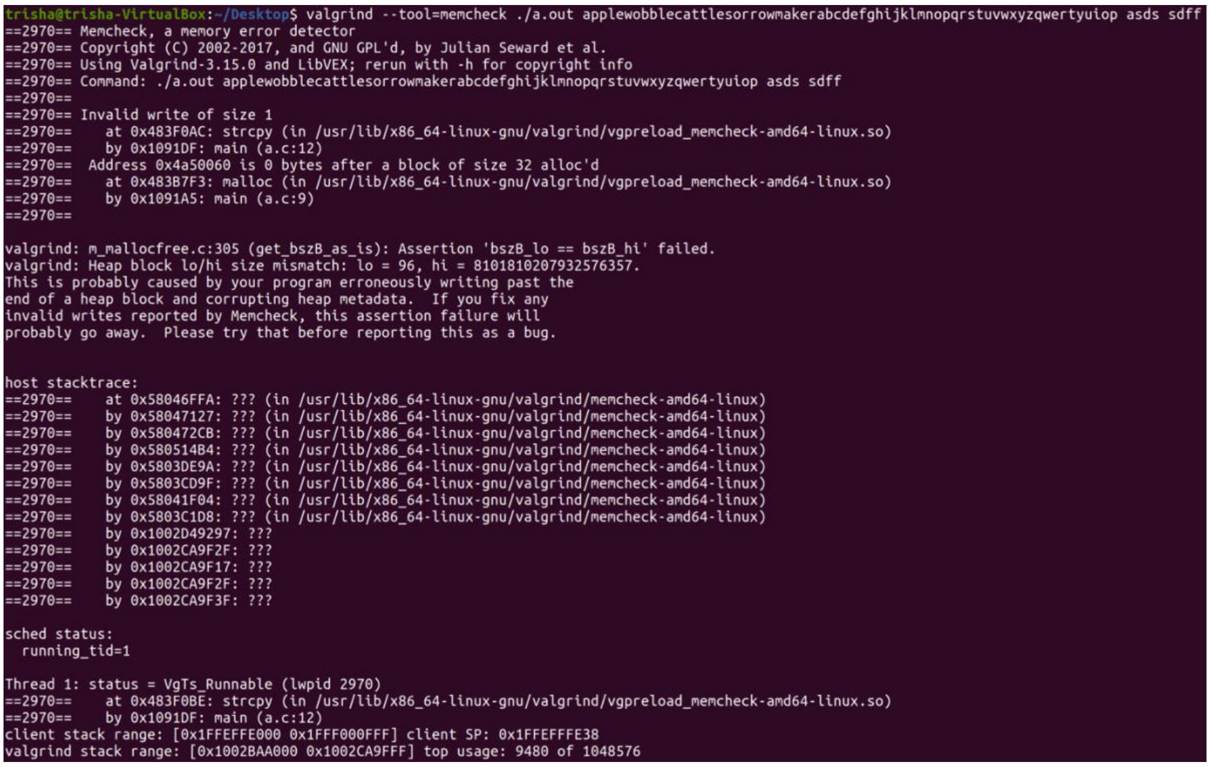
If arguments entered are within the allocated range :-



```
trisha@trisha-VirtualBox:~/Desktop$ valgrind --tool=memcheck ./a.out 10 20 30
==2945== Memcheck, a memory error detector
==2945== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2945== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2945== Command: ./a.out 10 20 30
==2945==
==2945==
==2945== HEAP SUMMARY:
==2945==     in use at exit: 0 bytes in 0 blocks
==2945==   total heap usage: 3 allocs, 3 frees, 96 bytes allocated
==2945==
==2945== All heap blocks were freed -- no leaks are possible
==2945==
==2945== For lists of detected and suppressed errors, rerun with: -s
==2945== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

<u>Observation</u>:- Since in this case the size of the arguments entered are less that the allocated block in the main function, there are no errors generated. *Valgrind reports that there were 3 allocs and 3 frees and no memory leaks are detected.*

If arguments entered are beyond the allocated range :-



```
trisha@trisha-VirtualBox:~/Desktop$ valgrind --tool=memcheck ./a.out applewobblecattlesorrowmakerabcdefghijklmnopqrstuvwxyzqwertyuiop asds sdff
==2970== Memcheck, a memory error detector
==2970== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==2970== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==2970== Command: ./a.out applewobblecattlesorrowmakerabcdefghijklmnopqrstuvwxyzqwertyuiop asds sdff
==2970==
==2970== Invalid write of size 1
==2970==    at 0x483F0AC: strcpy (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==2970==    by 0x1091DF: main (a.c:12)
==2970==  Address 0x4a50060 is 0 bytes after a block of size 32 alloc'd
==2970==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==2970==    by 0x1091A5: main (a.c:9)
==2970==

valgrind: m_mallocfree.c:305 (get_bszB_as_is): Assertion 'bszB_lo == bszB_hi' failed.
valgrind: Heap block lo/hi size mismatch: lo = 96, hi = 8101810207932576357.
This is probably caused by your program erroneously writing past the
end of a heap block and corrupting heap metadata.  If you fix any
invalid writes reported by Memcheck, this assertion failure will
probably go away.  Please try that before reporting this as a bug.


host stacktrace:
==2970==    at 0x58046FFA: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/memcheck-amd64-linux)
==2970==    by 0x58047127: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/memcheck-amd64-linux)
==2970==    by 0x580472CB: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/memcheck-amd64-linux)
==2970==    by 0x580514B4: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/memcheck-amd64-linux)
==2970==    by 0x5803DE9A: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/memcheck-amd64-linux)
==2970==    by 0x5803CD9F: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/memcheck-amd64-linux)
==2970==    by 0x58041F04: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/memcheck-amd64-linux)
==2970==    by 0x5803C1D8: ??? (in /usr/lib/x86_64-linux-gnu/valgrind/memcheck-amd64-linux)
==2970==    by 0x1002D49297: ???
==2970==    by 0x1002CA9F2F: ???
==2970==    by 0x1002CA9F17: ???
==2970==    by 0x1002CA9F2F: ???
==2970==    by 0x1002CA9F3F: ???

sched status:
  running_tid=1

Thread 1: status = VgTs_Runnable (lwpid 2970)
==2970==    at 0x483F0BE: strcpy (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==2970==    by 0x1091DF: main (a.c:12)
client stack range: [0x1FFEFFE000 0x1FFF000FFF] client SP: 0x1FFEFFFE38
valgrind stack range: [0x1002BAA000 0x1002CA9FFF] top usage: 9480 of 1048576
```

<u>Observation</u>:- In this case the first argument that is entered is more than 32 bytes and strcpy does not check the size of the string hence it leads to buffer overflow.

Valgrind's memcheck reports that the heap block has a lo/high mismatch which is probably caused by our program erroneously writing past the end of a heap block and corrupting the heap metadata. And it advises us to fix this invalid write.

Without Valgrind :

<u>In the above program the following errors lead to a vulnerable program :-</u>

1) In lines 5, 6, 7 the successful allocation of the blocks by malloc is not verified so if malloc fails then the program will not function as expected. The size of the memory to be allocated is hardcoded instead of using sizeof operator on char, which can also lead to implementation defined errors.

2) In lined 8, 9, 10 strcpy function is used which does not check the size of the command line arguments and thus if the arguments entered are more than 32 bytes then it will result in buffer overflow.

1b) Review the below code:

i) Does it lead to memory leak?

ii) If so, correct it to avoid vulnerability?

Code:

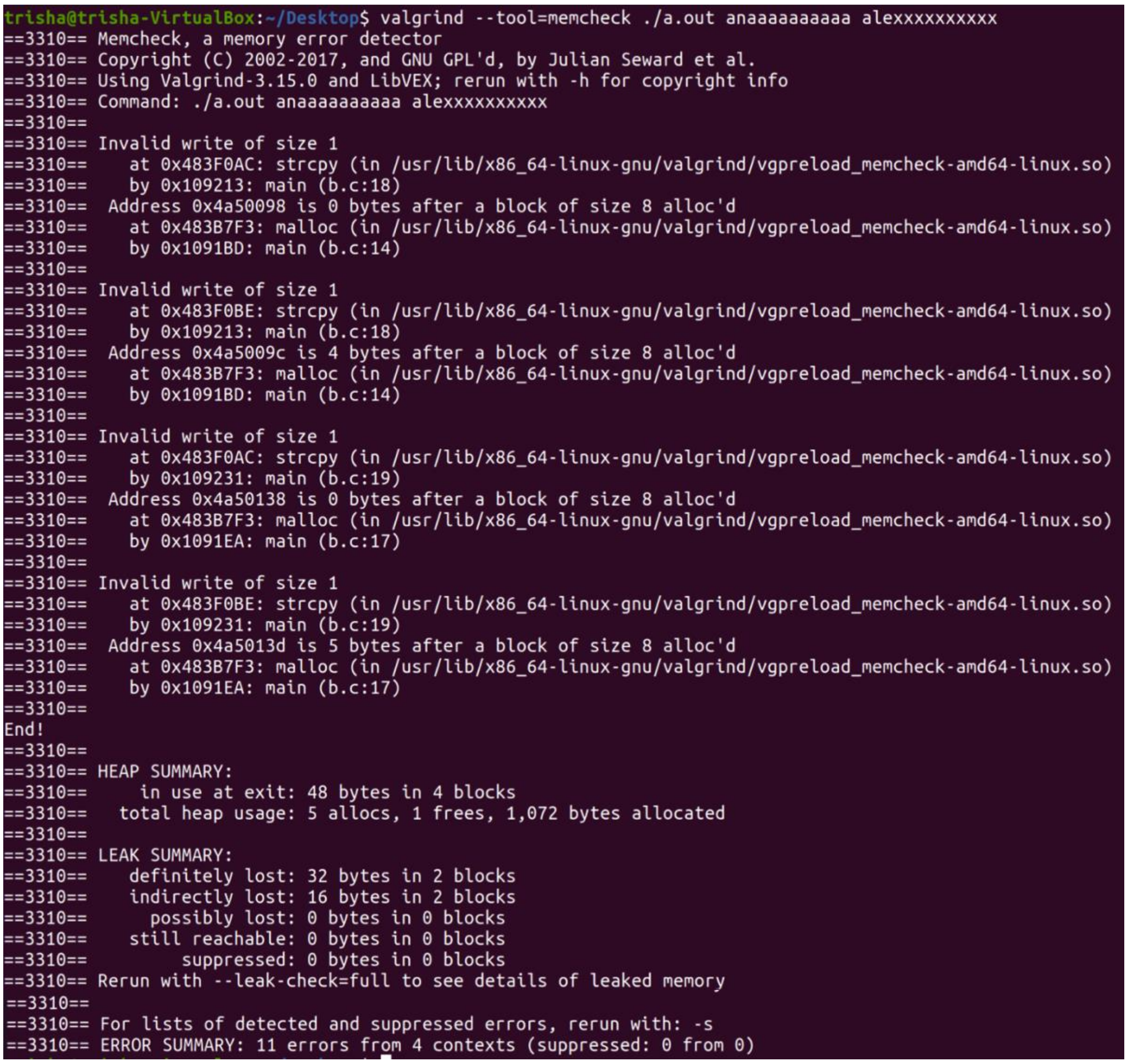
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Sample {
        int priority;
        char *name;
};
int main(int argc, char **argv) {
        struct Sample *i1;
        struct Sample *i2;
        i1 = malloc(sizeof(struct Sample));
        i1->priority = 1;
        i1->name = malloc(8);
        i2 = malloc(sizeof(struct Sample));
        i2->priority = 2;
        i2->name = malloc(8);
        strcpy(i1->name, argv[1]);
        strcpy(i2->name, argv[2]);
        printf("End!\n");
        return 0;
}
```

<u>Solution :-</u>

(i) Yes, it leads to memory leak.
This code leads to memory leak because it does not free the dynamically allocated memory at the end of the program.
It also does not check the size of the entered command line arguments as it used strcpy function, which could lead to buffer overflow. It also does not check the successful allocation of memory using malloc.
And while allocating the memory dynamically it hardcodes the value instead of using the sizeof operator.

Valgrind ScreenShot :-

According to the valgrind output there are invalid write errors because we entered a command line argument of size greater than 8 bytes which leads to buffer overflow. And it also indicates that memory is lost directly because of the memory allocated for the elements of the struct Sample and indirectly lost because of the memory allocated for char* name in the struct Sample.



```
trisha@trisha-VirtualBox:~/Desktop$ valgrind --tool=memcheck ./a.out anaaaaaaaaaa alexxxxxxxxxx
==3310== Memcheck, a memory error detector
==3310== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3310== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3310== Command: ./a.out anaaaaaaaaaa alexxxxxxxxxx
==3310==
==3310== Invalid write of size 1
==3310==    at 0x483F0AC: strcpy (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==3310==    by 0x109213: main (b.c:18)
==3310==  Address 0x4a50098 is 0 bytes after a block of size 8 alloc'd
==3310==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==3310==    by 0x1091BD: main (b.c:14)
==3310==
==3310== Invalid write of size 1
==3310==    at 0x483F0BE: strcpy (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==3310==    by 0x109213: main (b.c:18)
==3310==  Address 0x4a5009c is 4 bytes after a block of size 8 alloc'd
==3310==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==3310==    by 0x1091BD: main (b.c:14)
==3310==
==3310== Invalid write of size 1
==3310==    at 0x483F0AC: strcpy (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==3310==    by 0x109231: main (b.c:19)
==3310==  Address 0x4a50138 is 0 bytes after a block of size 8 alloc'd
==3310==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==3310==    by 0x1091EA: main (b.c:17)
==3310==
==3310== Invalid write of size 1
==3310==    at 0x483F0BE: strcpy (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==3310==    by 0x109231: main (b.c:19)
==3310==  Address 0x4a5013d is 5 bytes after a block of size 8 alloc'd
==3310==    at 0x483B7F3: malloc (in /usr/lib/x86_64-linux-gnu/valgrind/vgpreload_memcheck-amd64-linux.so)
==3310==    by 0x1091EA: main (b.c:17)
==3310==
End!
==3310==
==3310== HEAP SUMMARY:
==3310==     in use at exit: 48 bytes in 4 blocks
==3310==   total heap usage: 5 allocs, 1 frees, 1,072 bytes allocated
==3310==
==3310== LEAK SUMMARY:
==3310==    definitely lost: 32 bytes in 2 blocks
==3310==    indirectly lost: 16 bytes in 2 blocks
==3310==      possibly lost: 0 bytes in 0 blocks
==3310==    still reachable: 0 bytes in 0 blocks
==3310==         suppressed: 0 bytes in 0 blocks
==3310== Rerun with --leak-check=full to see details of leaked memory
==3310==
==3310== For lists of detected and suppressed errors, rerun with: -s
==3310== ERROR SUMMARY: 11 errors from 4 contexts (suppressed: 0 from 0)
```

(ii) Corrected Code :-

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Sample {
        int priority;
        char *name;
};
int main(int argc, char **argv) {
        struct Sample *i1;
        struct Sample *i2;
        i1 = malloc(sizeof(struct Sample));
        if(i1 == NULL){                                    //checking allocation
                printf("Memory not allocated successfully\n");
                return 0;
        }
        i1->priority = 1;
        i1->name = malloc((sizeof(char))*8);          //using sizeof operator
        if(i1->name = NULL){                               //checking allocation
                printf("Memory not allocated successfully\n");
                return 0;
        }
        i2 = malloc(sizeof(struct Sample));
        if(i2 == NULL){                                     //checking allocation
                printf("Memory not allocated successfully\n");
                return 0;
        }
        i2->priority = 2;
        i2->name = malloc(sizeof(char)*8);            //using sizeof operator
        if(i2->name == NULL){                              //checking allocation
                printf("Memory not allocated successfully\n");
                return 0;
        }
        strncpy(i1->name, argv[1], sizeof(char)*8);   //using strncpy
        strncpy(i2->name, argv[2], sizeof(char)*8);   //using strncpy
        printf("End!\n");
        free(i1->name);             //freeing memory
        free(i1);                   //freeing memory
        free(i2->name);             //freeing memory
        free(i2);                   //freeing memory
        return 0;
}
```

Question2:

2a) Given below is the non-compliant code snippet:

i) Identify the line (or instruction) which may lead to exploit.

ii) Write compliant solution (or code snippet) for the same to avoid the exploit.

Code:

```
void foo2(int *list, size_t list_size) {
    if (size < MIN_SIZE) {
    free(list);
    return; }
```

```
          /* Continue to process the list */
 }
 //...
void foo1 (size_t number) {
        int *list = malloc(number * sizeof(int));
        if (list == NULL) {
                /* Handle Allocation Error */
        }
        foo2(list, number);
        /* Continue Processing list */
        free(list);
}
 //...
```

<u>Solution :-</u>

(i) If the number passed as an argument to the foo1 function is less than MIN_SIZE, then the foo2 function will free the list passed as an argument to the foo2 function. Then it returns the control to the function that called it which is foo1.
When foo1 gets the control back, it has to process the list but this will not be possible because the list is already freed. Then the foo1 function proceeds to free the list which is already freed by the foo2 function.
So, this means the free(list) instruction will be called twice and this will result in undefined behaviour.
Thus this can lead to exploit of data because if the foo2 function frees the list then the list pointer will be a dangling pointer which could be exploited.

(ii) <u>Compliant Solution :-</u>

```
void foo2(int *list, size_t list_size) {
        if (size < MIN_SIZE) {    //checks the size argument
        free(list);          //frees the list
        list = NULL;      //to avoid dangling pointer error
        return; }         //returns back to the callee function
         /* Continue to process the list */
 }
 //...
void foo1 (size_t number) {
        int *list = malloc(number * sizeof(int));   //allocates ‘number’ bytes of int
        if (list == NULL) {
                /* Handle Allocation Error */
        }
        foo2(list, number);        //calls the foo2 function with list and ‘number’
        if(list == NULL)                        //to avoid double freeing
                return;
        /* Continue Processing list */
        free(list);            //frees list if not already freed by foo2
}
 //...
```

2b) Understand the code given below to answer these questions:

i) What vulnerability does it contain that might lead to modify program execution?

ii) How can you modify this code to mitigate the vulnerability, if exists?

Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int x;
void disp(char *str) {
        printf(str);
}
int main(int argc, char **argv) {
        char buf[256];
        if(argc>1) {
                memset(buf,0,sizeof(buf));
                strncpy(buf,argv[1],sizeof(buf));
                disp(buf);
        }
        if(x!=0) {
                printf("x variable has been changed correctly!");
        }
        else {
        printf("Hello all, you didn't succeed\n");
        }
        return 0;
}
```

(i) The command line argument that is entered is a tainted data and it is not sanitized this could cause vulnerability which can lead to modification of program execution.
In the disp function the printf instruction does not have a format specifier so this could also lead to vulnerabilities because the arguments cannot be type checked.

(ii) The program can be modified to ensure sanitization of the entered command line argument to mitigate the vulnerability. Sanitization can be performed by either termination or replacement.
The format specifier for the printf statement must be explicitly specified to avoid vulnerabilities.
printf("%s", str);