DEVICE DRIVERS – LAB EXERCISE 8

Submitted By:

Kiran Thomas Cherian

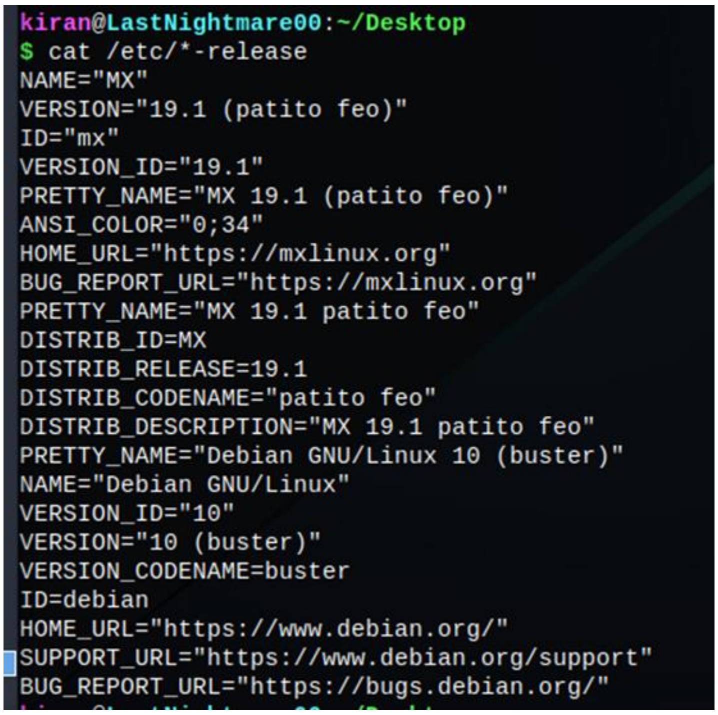
CED18I028

OBJECTIVE:

Write a C program to get a segment fault error.

Linux Distribution Used:

MX Linux 19.1 (Running on Virtual machine)



```
kiran@LastNightmare00:~/Desktop
$ cat /etc/*-release
NAME="MX"
VERSION="19.1 (patito feo)"
ID="mx"
VERSION_ID="19.1"
PRETTY_NAME="MX 19.1 (patito feo)"
ANSI_COLOR="0;34"
HOME_URL="https://mxlinux.org"
BUG_REPORT_URL="https://mxlinux.org"
PRETTY_NAME="MX 19.1 patito feo"
DISTRIB_ID=MX
DISTRIB_RELEASE=19.1
DISTRIB_CODENAME="patito feo"
DISTRIB_DESCRIPTION="MX 19.1 patito feo"
PRETTY_NAME="Debian GNU/Linux 10 (buster)"
NAME="Debian GNU/Linux"
VERSION_ID="10"
VERSION="10 (buster)"
VERSION_CODENAME=buster
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"
```

A segmentation fault occurs when a program attempts to access a memory location that it is not allowed to access, or attempts to access a memory location in a way that is not allowed (for example, attempting to write to a read-only location, or to overwrite part of the operating system).

The following are some typical causes of a segmentation fault:

- Attempting to access a non-existent memory address (outside process's address space)
- Attempting to access memory the program does not have rights to (such as kernel structures in process context)
- Attempting to write read-only memory (such as code segment)

These in turn are often caused by programming errors that result in invalid memory access:

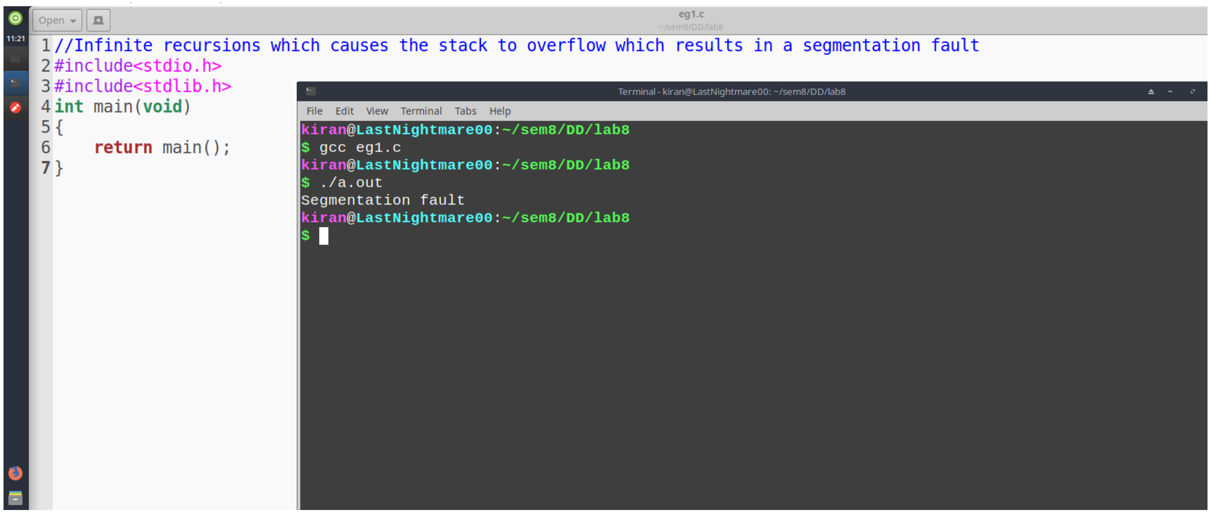
- Dereferencing a null pointer, which usually points to an address that's not part of the process's address space
- Dereferencing or assigning to an uninitialized pointer (wild pointer, which points to a random memory address)
- Dereferencing or assigning to a freed pointer (dangling pointer, which points to memory that has been freed/deallocated/deleted)
- A buffer overflow
- A stack overflow
- Attempting to execute a program that does not compile correctly.

EXAMPLE 1

CODE:

```
//Infinite recursions which causes the stack to overflow which results in a segmentation fault

#include<stdio.h>

#include<stdlib.h>

int main(void)

{

   return main();

}
```

OUTPUT SCREENSHOT:


Open
eg1.c
~/sem8/DD/lab8
11:21
Terminal - kiran@LastNightmare00: ~/sem8/DD/lab8
File Edit View Terminal Tabs Help

```
//Infinite recursions which causes the stack to overflow which results in a segmentation fault
#include<stdio.h>
#include<stdlib.h>
int main(void)
{
    return main();
}
```

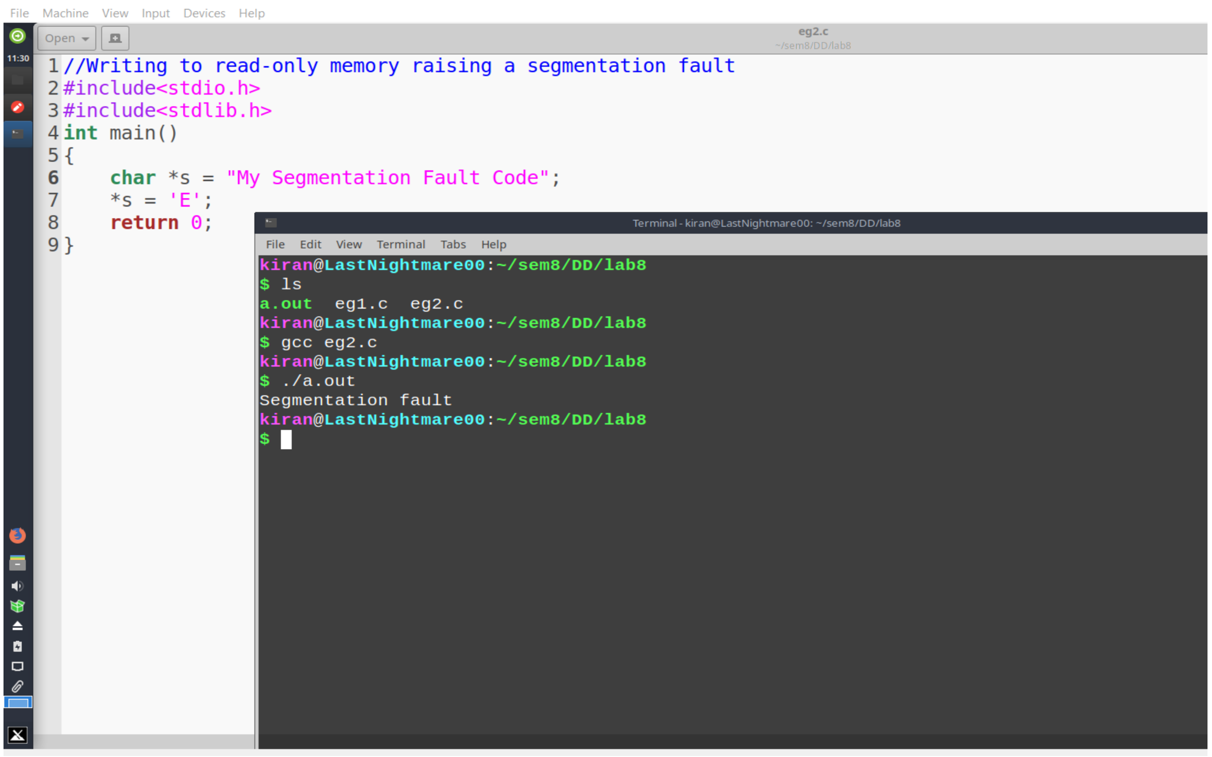
```
kiran@LastNightmare00:~/sem8/DD/lab8
$ gcc eg1.c
kiran@LastNightmare00:~/sem8/DD/lab8
$ ./a.out
Segmentation fault
kiran@LastNightmare00:~/sem8/DD/lab8
$ 
```

EXAMPLE 2

CODE:

```
//Writing to read-only memory raising a segmentation fault
#include<stdio.h>
#include<stdlib.h>
int main()
{
   char *s = "My Segmentation Fault Code";
   *s = 'E';
        return 0;
}
```

Output Screenshot:


File Machine View Input Devices Help
Open
eg2.c
~/sem8/DD/lab8
Terminal - kiran@LastNightmare00: ~/sem8/DD/lab8
File Edit View Terminal Tabs Help

```
//Writing to read-only memory raising a segmentation fault
#include<stdio.h>
#include<stdlib.h>
int main()
{
    char *s = "My Segmentation Fault Code";
    *s = 'E';
    return 0;
}
```

```
kiran@LastNightmare00:~/sem8/DD/lab8
$ ls
a.out  eg1.c  eg2.c
kiran@LastNightmare00:~/sem8/DD/lab8
$ gcc eg2.c
kiran@LastNightmare00:~/sem8/DD/lab8
$ ./a.out
Segmentation fault
kiran@LastNightmare00:~/sem8/DD/lab8
$ 
```