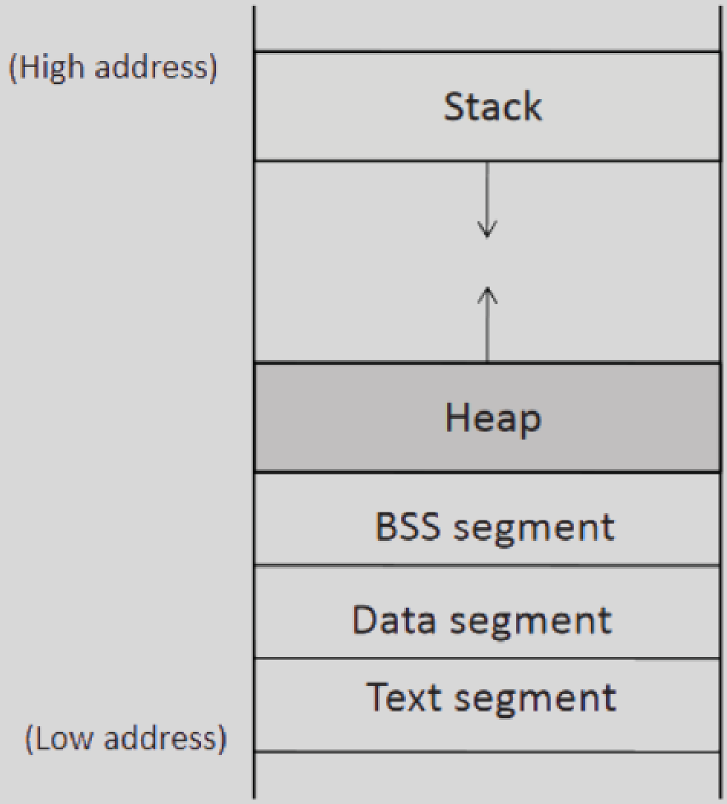
Buffer Overflow on the Programmer Perspective

Before learning Buffer Overflow vulnerability, we need to learn the working principles of the concept of Memory. When a program runs, it needs a certain amount of memory. A typical C program divides the memory into five different segments (parts), and each piece serves a purpose. The five parts that are divided are called Program Memory Layout. The figure below shows the order of the specified parts.



The Low address and High Address: show us which Program Memory Layout is among the values.

Stack: Contains the variables defined in the program.

Heap: Used to create Dynamic Memory Distribution. It is processed by commands such as Malloc, Calloc, Realloc, and Free.

BSS: It is used to store Static and Dynamic variables that are not used yet. If its content is not yet in use, it is filled with 0 (zero).

Data: Used to store the Static and Dynamic variables used.

Text: Contains executable program codes. This section is generally only readable.

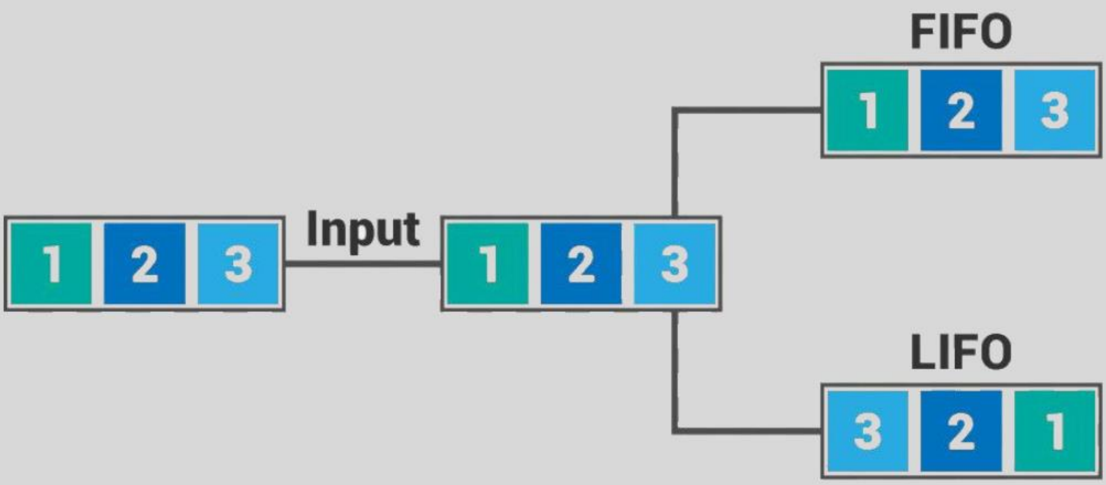
Now let's look at the working principles of the specified segments with the help of codes;

```
##### Ethical Hacking - Sam Houston State University #####
#include <studio.h>
#include <limits.h>
#include <float.h>
int Global_Variable = 100;
int main()
{
        #### Content to be stored in Stack Segment ####
        int number_1 = 1 ;
        float number_2 = 2.5 ;
        static int Static_Variable;
        #### Defining Memory for the Heap Segment ####

        int*ptr = (int*) malloc(2*sizeof(int));
        #### Content to be stored in Heap Segment ####
        ptr[0]=5;
        ptr[1]=6;
        #### Releasing Memory Defined for the Heap Segment ####
        free(ptr);
        return 1;
}
```

It is the Stack Segment that we will examine among the specified Segments. Now let's touch the Stack structure in more detail.

Stack

In Computer Science, Abstract Data Type is the name given to the structure that regulates the operations on the data. One of the most famous elements of the Abstract Data Type structure is the Stack concept. Stack data type works with Last in First Out (LIFO) logic.



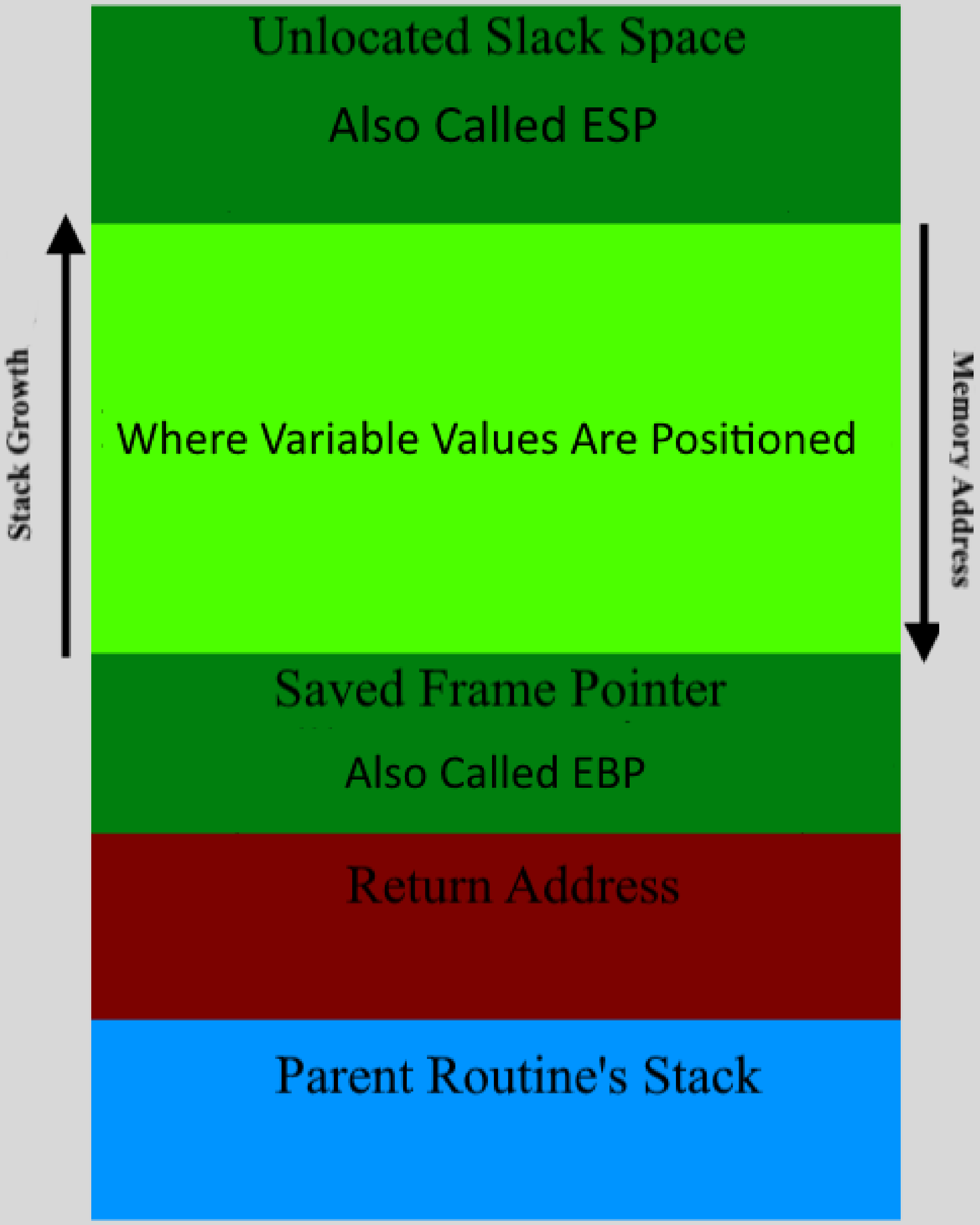
As can be easily understood in the image, although the number 3 is added to the last row, it will be the first output with the LIFO (Last in First Out) logic. Stack variable has three different functions;

Push → Adds data to the Stack (First Place)

Pop → Receives data from the stack (From First Place)

Top → Retrieves the first data from Stack but does not delete the data.

Stack Memory Layout



When the variables reach the Stack Segment by the program, the Memory Address will show down as the addition process works with LIFO logic. As the data will be read, Stack Growth will point Upwards as the Last Added is the first to be read.

Unallocated Slack Space (ESP): Not available for use unless defined by the programmer. The added variables and their contents will come to the light green area. It usually helps to show Last In or First Out data.

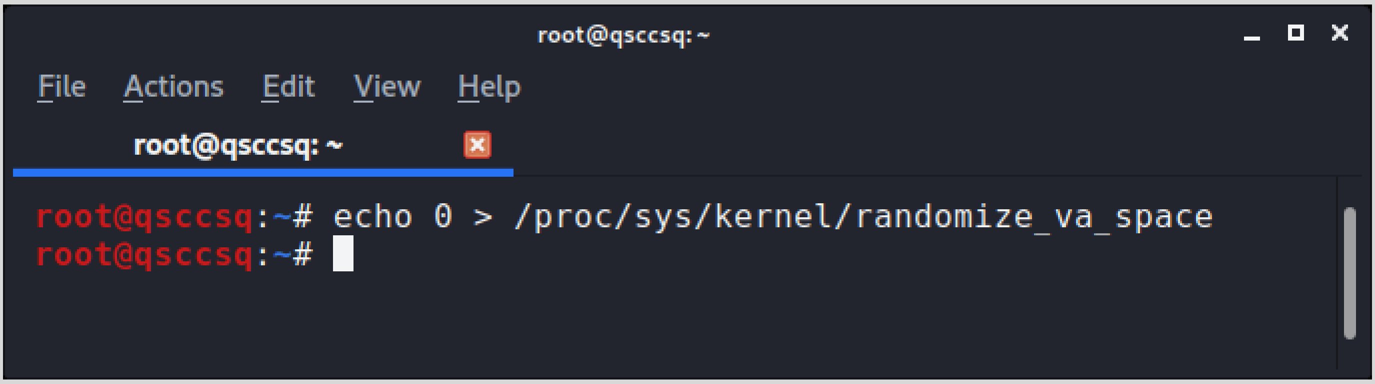
Saved Frame Pointer (EBP): Shows First In or Last Out Data.

Return Address: It shows the returns in the code. It can be thought like a For Loop. Does not finish his work without returning the given value.

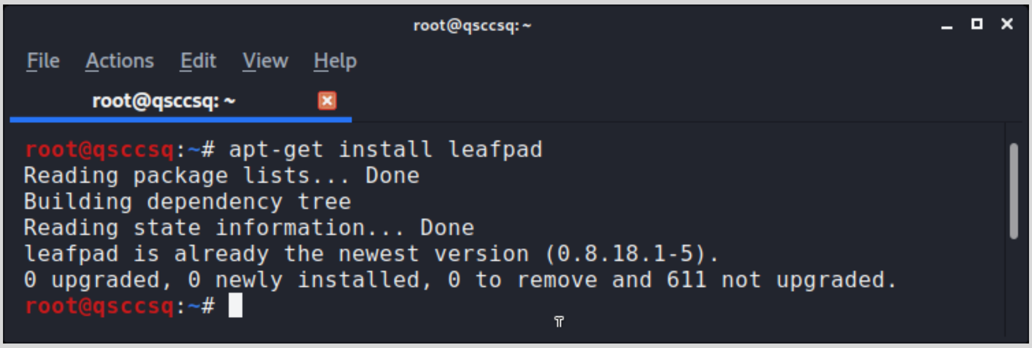
Parent Routine's Stack: Identifies and processes addresses registered by the CPU.

Let's demonstrate with an example in order to better understand the information provided.

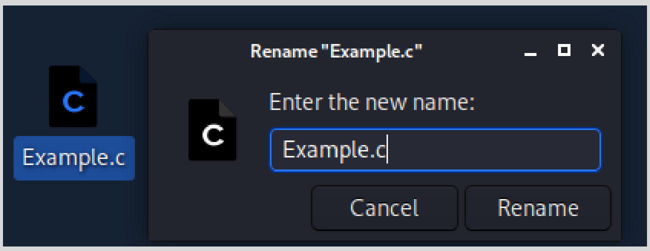
First Step : Type "echo 0 > /proc/sys/kernel/randomize_va_space" to Kali Linux terminal;



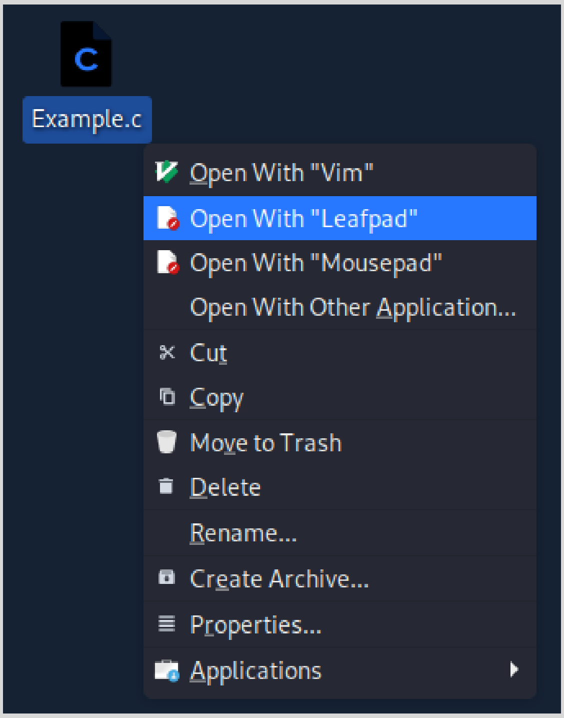
Second Step : Type "apt-get install leafpad" to Kali Linux terminal;



Third Step: Create an empty document to desktop (Example.c);

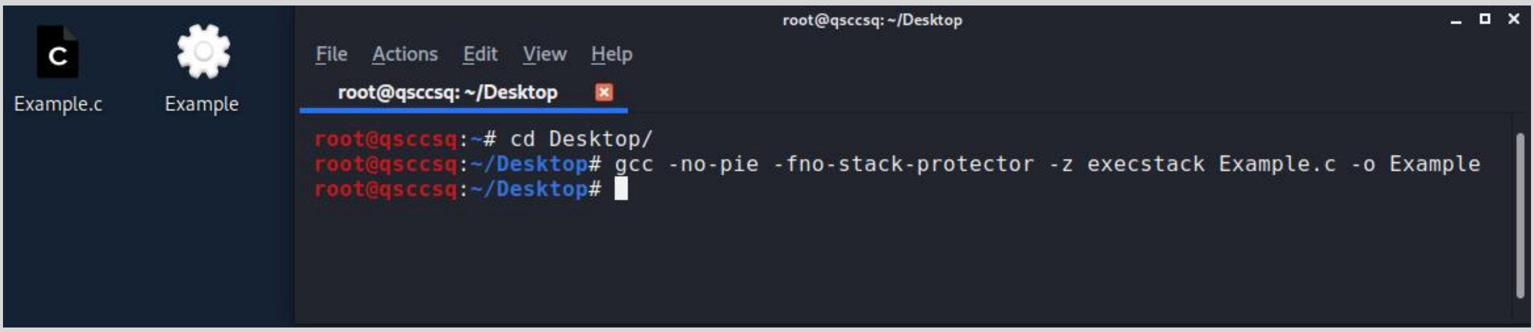


Fourth Step: Open Example.c document with Leafpad and fill it up as in the image;

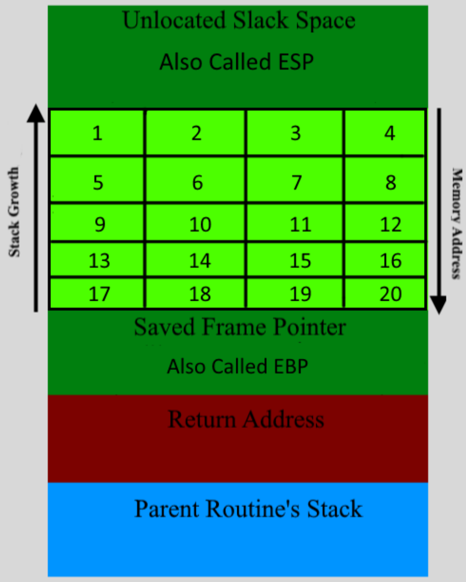


Fifth Step: Type “cd Desktop” and let’s turn our example document to executiable format with terminal;

Code : gcc -no-pie -fno-stack-protector -z execstack Example.c -o Example

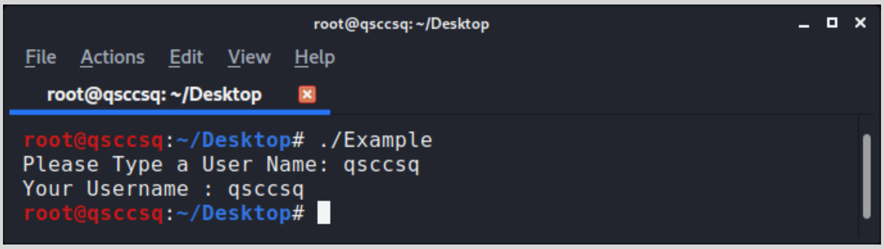


Username in the 4th line of the code given above can take up to 20 characters. The state of Stack Segment will be as follows;

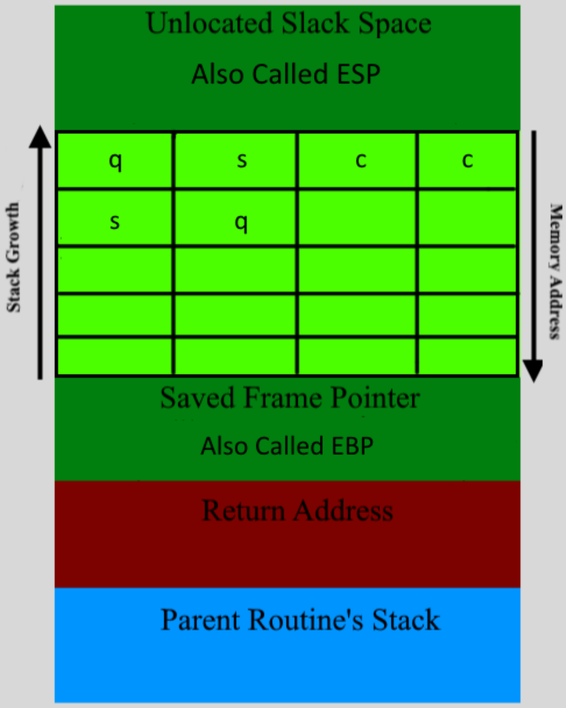


As you see at above, we already have 20 space to input data. Let's we check what happen when we put some info on it;

Go back to Kali's terminal and type "./Example.c" then type an username;



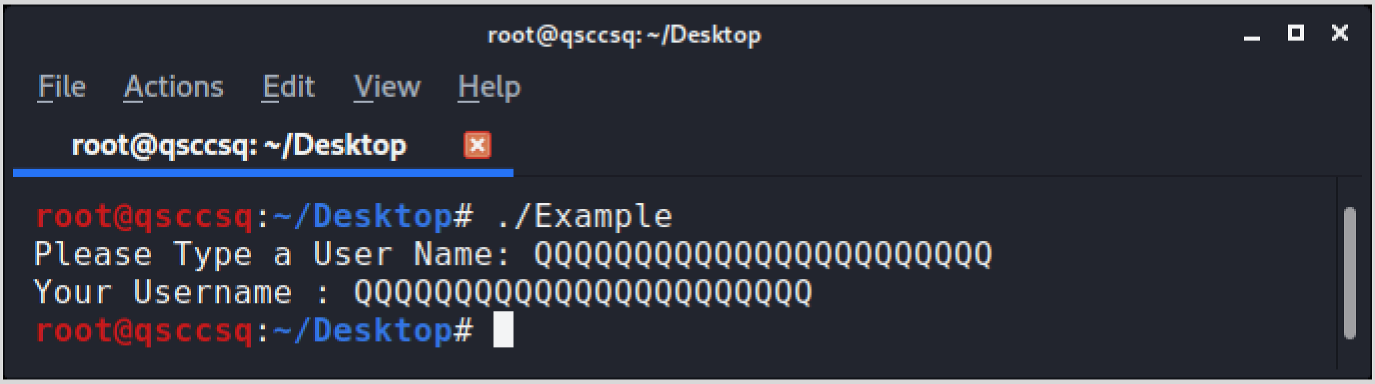
The state of our Stack Segment will be as follows;

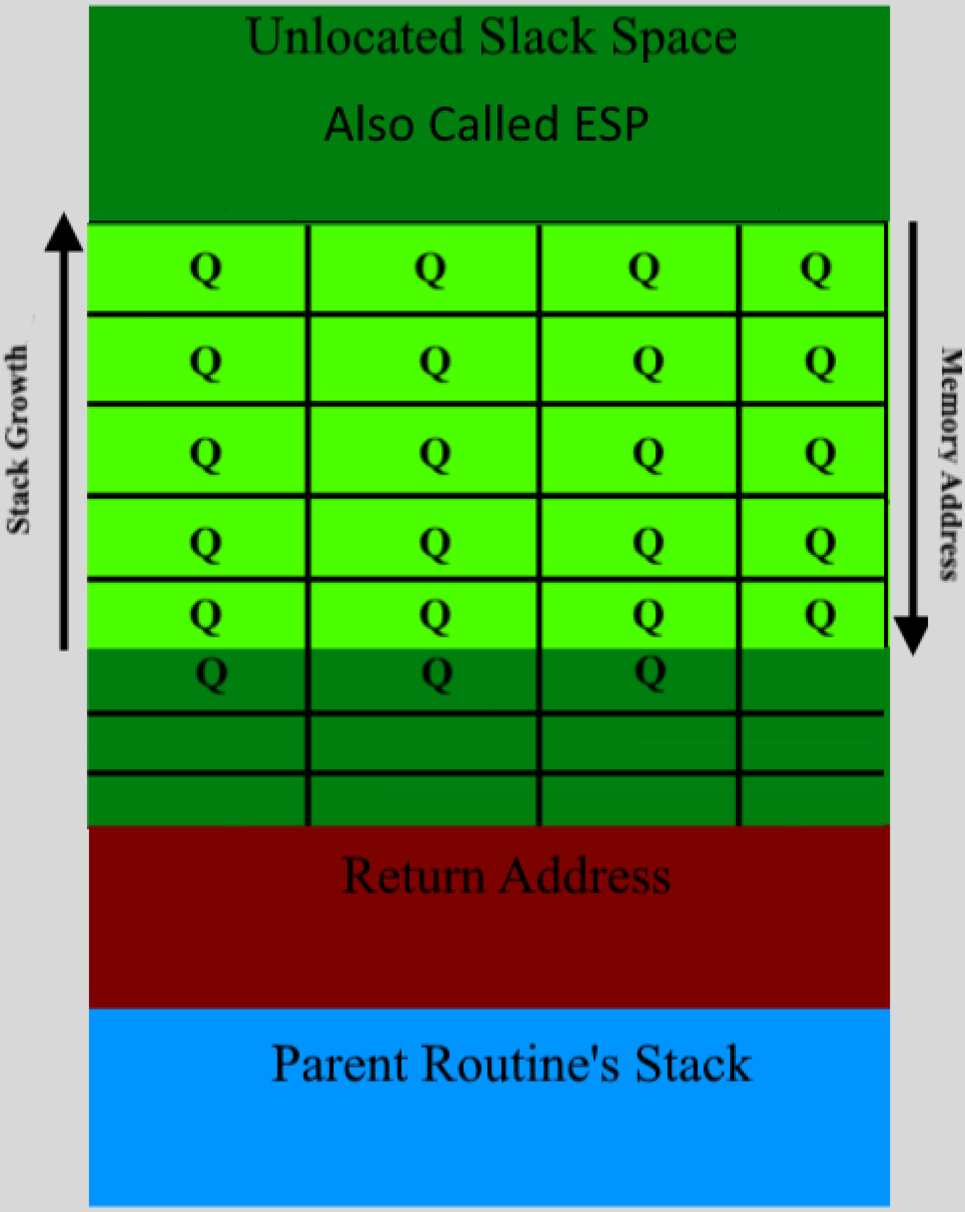


Buffer Overflow on the Hacker Perspective

We have seen a detailed review of a simple programming above. So how are these types of programs abused by Hackers? As you may remember, we had a 20-character field. So what happens if we cross borders?

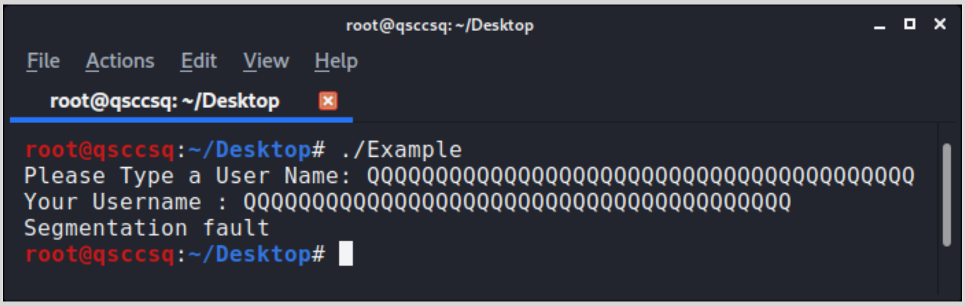
Now let's enter 23 characters and observe the results;



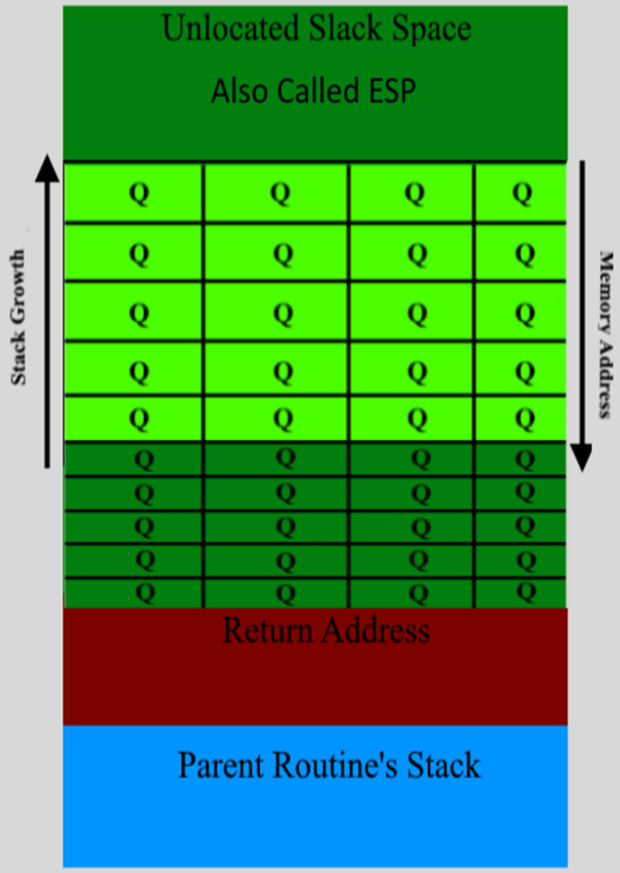


As you can see, Saved Frame Pointer contains areas to hide characters in itself. In addition to the 20 previously registered characters, there is a field in the Saved Frame Pointer where we can add an unknown number of characters. Now our goal is to control how many data can be saved by the Saved Frame Pointer.

We need to check how many data can be saved with the Saved Frame Pointer (EBP) by trial method until we get an error. As a result of my individual attempts, I came up to the 39th character and when I entered the 40th character, I finally encountered an error.



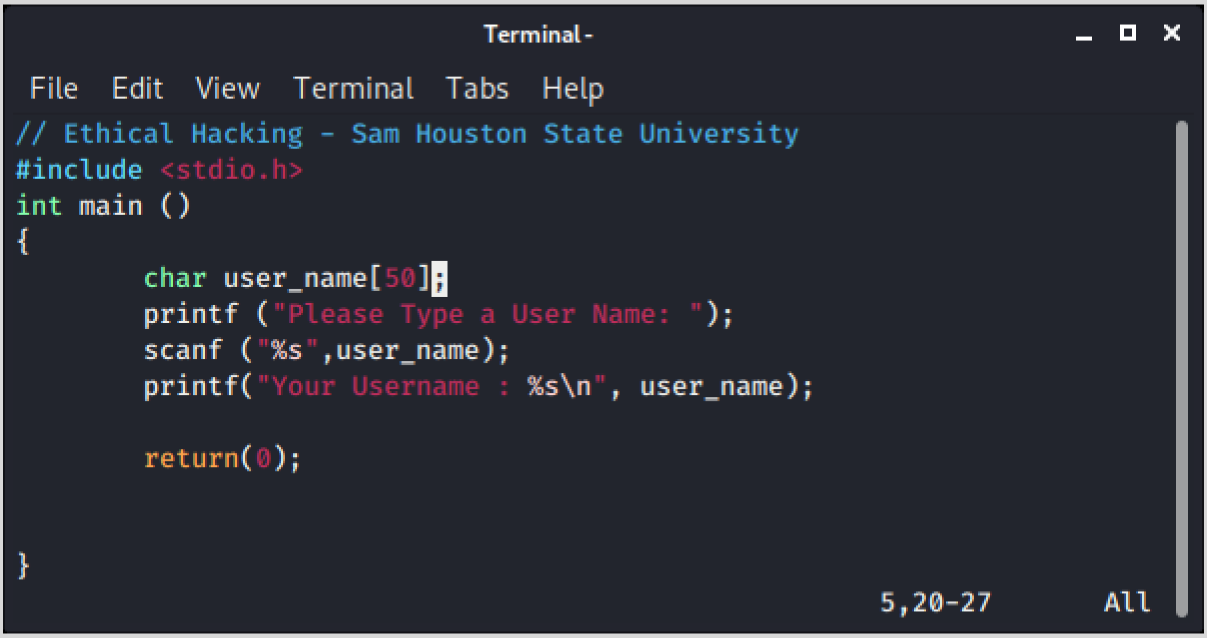
Let's check the state of our Stack Segment;



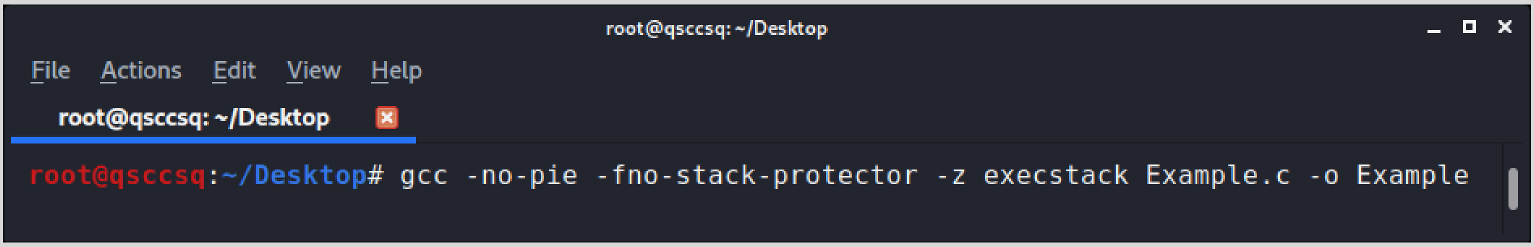
The Return Segment stopped the program because it could not get the variable values. Our last letter "Q" did not fit into memory and started to overflow. This is exactly the Buffer Overflow, or Stack overflow.

Creating Fuzzer to Make Things Easier

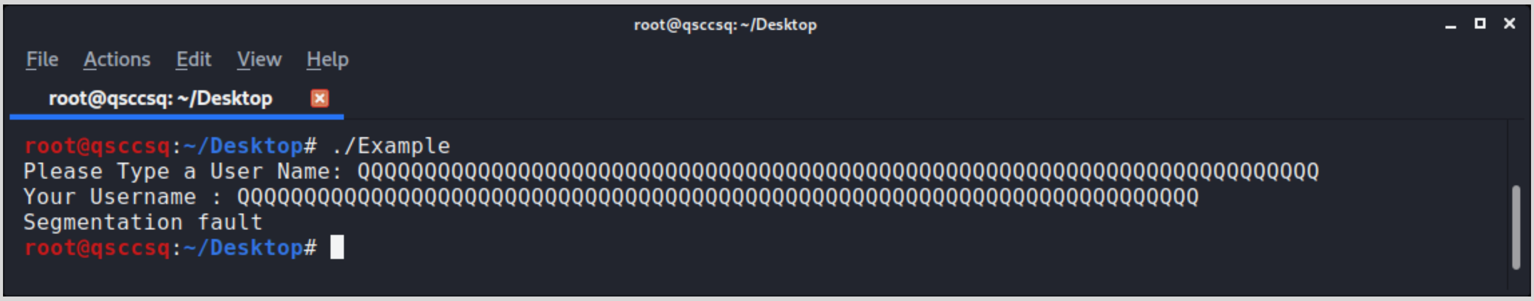
Based on the example above, it can be considered that we receive an error message if twice the limit of the application is entered. However, the analysis is completely wrong. Now let's set the input limit of our application to 50, then let's see how many values enough to get segmentation fault.



Compile it with "gcc -no-pie -fno-stack-protector -z execstack Example.c -o Example"



Let's put values to application;



In our previous conclusion, we thought that when we entered 100 data, we could get an error, but 72 values were sufficient. So what happens if we make our value 500? If you want, let's write an application to facilitate these operations. Python language is the most used programming language in Exploit writing thanks to its ease. Fuzzing or Fuzz is the name given to the exploits written to automate the processes. Let's write a Buffer Overflow Detector Fuzzer for our application using Python language. Our aim will be to find out exactly what value we receive the error message by creating the data via Python and sending it to our target application.

Extra Point: Analyze the creation steps of the application, whose codes are shown below, and explain at least half a page (30 Points). NOTE: BEST WAY IS CREATE SAME APPS IN YOUR PC THEN RUN IT STEP BY STEP. If you have not Python-IDLE type "apt-get install idle-python3.7" to terminal then create a document to your desktop with ".py" extension. Finally, right click on it and click on open with another application. You'll see IDLE-PYTHON on the list.

Fuzzer.py - /root/Desktop/Fuzzer.py (3.7.7)

File Edit Format Run Options Window Help

```
#!/usr/bin/python
import os
import subprocess

if os.path.isfile("/root/Desktop/counter.txt"):
    file1 = open("/root/Desktop/counter.txt","r+")
    from_file = int(file1.read())
    print ('s' * from_file)
    from_file = from_file + 1
    file1 = open("/root/Desktop/counter.txt","w")
    file1.write(str(from_file))
    file1.close()



else:
    file1 = open("/root/Desktop/counter.txt","w")
    file1.write("1")
    file1.close()
```

Ln: 1 Col: 0

Runner.py - /root/Desktop/Runner.py (3.7.7)

File Edit Format Run Options Window Help

```
import os
import subprocess

while (1):
    os.system('/root/Desktop/Fuzzer.py')


    subprocess.Popen("./Fuzzer.py > f", shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE).communicate()[0]
    p = subprocess.Popen("./Example < f", shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE).communicate()[0]

    if (len(p)) < 5:
        file1 = open("/root/Desktop/counter.txt","r+")
        from_file = int(file1.read())
        exact_number = from_file - 2
        print("Target Application Has BufferOverflow with",str(exact_number),"Values !")
        os.remove("/root/Desktop/counter.txt")
        os.remove("/root/Desktop/f")

        break
```

Ln: 1 Col: 0

Now, to test our application, let's make our limit 500 and run our application.

```
Terminal -
File  Edit  View  Terminal  Tabs  Help
// Ethical Hacking - Sam Houston State University
#include <stdio.h>
int main ()
{
        char user_name[500];
        printf ("Please Type a User Name: ");
        scanf ("%s",user_name);
        printf("Your Username : %s\n", user_name);

        return(0);
                                                5,21-28        Top
```

Compile it;

```
root@qsccsq: ~/Desktop
File  Actions  Edit  View  Help
root@qsccsq: ~/Desktop
root@qsccsq:~/Desktop# gcc -no-pie -fno-stack-protector -z execstack Example.c -o Example
root@qsccsq:~/Desktop# 
```

Run Runney.py

```
Runner.py - /root/Desktop/Runner.py (3.7.7)
File  Edit  Format  Run  Options  Window  Help
import os
import subprocess

while (1):
    os.system('/root/Desktop/Fuzzer.py')


    subprocess.Popen("./Fuzzer.py > f", shell=True, stdout=subprocess.PIPE, stder
    p = subprocess.Popen("./Example < f", shell=True, stdout=subprocess.PIPE, std

    if (len(p)) < 5:
        file1 = open("/root/Desktop/counter.txt","r+")
        from_file = int(file1.read())
        exact_number = from_file - 2
        print("Target Application Has BufferOverflow with",str(exact_number),"Val
        os.remove("/root/Desktop/counter.txt")
        os.remove("/root/Desktop/f")

        break
```

```
Python 3.7.7 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.7.7 (default, Apr  1 2020, 13:48:52)
[GCC 9.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>>
====================== RESTART: /root/Desktop/Runner.py ======================
Target Application Has BufferOverflow with 520 Values !
>>>
```

Using GNU Debugger (GDB) to Understand What's Going on Our Application

During the development of our applications, there may be interruptions in the operation of our application due to some signals or interruptions or errors that come from the system or by the software developer. It may not be enough to be able to predict such situations most of the time. In such cases, our biggest helper will be GDB. An application called GBU debugger is commonly used in Linux systems. With this application, your application's code or core file can be examined. Let's change our limit to 20 again and examine our target application with GDB;

Installation of GDB;

```
root@qsccsq: ~
File  Actions  Edit  View  Help
root@qsccsq: ~

root@qsccsq:~# apt-get install gdb
Reading package lists... Done
Building dependency tree
Reading state information... Done
gdb is already the newest version (9.2-1).
```

Running Target Application with GDB;

```
root@qsccsq: ~/Desktop
File  Actions  Edit  View  Help
root@qsccsq: ~/Desktop

root@qsccsq:~/Desktop# gdb ./Example
GNU gdb (Debian 9.2-1) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./Example...
--Type <RET> for more, q to quit, c to continue without paging--c
(No debugging symbols found in ./Example)
(gdb)
```

Press "c" then ENTER

Run Target Application with type "run";

```
--Type <RET> for more, q to quit, c to continue without paging--c
(No debugging symbols found in ./Example)
(gdb) run
Starting program: /root/Desktop/Example
Please Type a User Name: C
Your Username : C
[Inferior 1 (process 8768) exited normally]
(gdb)
```

Run Application

I put C value. Exited Normally.

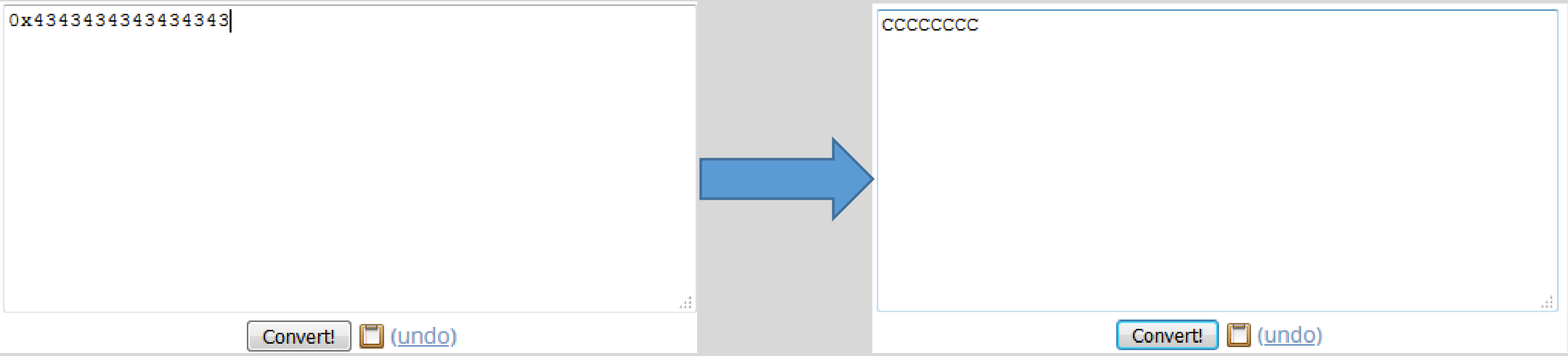
Run Target Application again and let we put 40 value on it;

```
(gdb) run
Starting program: /root/Desktop/Example
Please Type a User Name: CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
Your Username : CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7e1de02 in __libc_start_main (main=0x401132 <main>, argc=1,
    argv=0x7fffffffe298, init=<optimized out>, fini=<optimized out>,
    rtld_fini=<optimized out>, stack_end=0x7fffffffe288)
    at ../csu/libc-start.c:308
308     ../csu/libc-start.c: No such file or directory.
(gdb) 
```

We got segmentation fault. Let we look it deeply. So, let we disassemble our registers with "info registers" code;

```
root@qsccsq: ~/Desktop
File  Actions  Edit  View  Help
root@qsccsq: ~/Desktop

(gdb) info registers
rax            0x48                72
rbx            0x0                 0
rcx            0x0                 0
rdx            0x0                 0
rsi            0x4052a0            4215456
rdi            0x7ffff7fb64c0      140737353835712
rbp            0x4343434343434343  0x4343434343434343
rsp            0x7fffffffe1c0      0x7fffffffe1c0
r8             0xc100              49408
r9             0x39                57
r10            0x7fffffffe190      140737488347536
r11            0x246               582
r12            0x401050            4198480
r13            0x7fffffffe290      140737488347792
r14            0x0                 0
r15            0x0                 0
rip            0x7ffff7e1de02      0x7ffff7e1de02 <__libc_start_main+226>
eflags         0x10206             [ PF IF RF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) 
```

As you see, we have a constantly repeating set of values. Let put those values to Hexadecimal to Text converter.



Converter Website: https://www.browserling.com/tools/hex-to-text

RBP is used to deduce the program crashes by storing the last image of the stack pointer (RSP). In our example, we found that the RBP content overflowed with the "C" values we entered.

Debugging Target Application

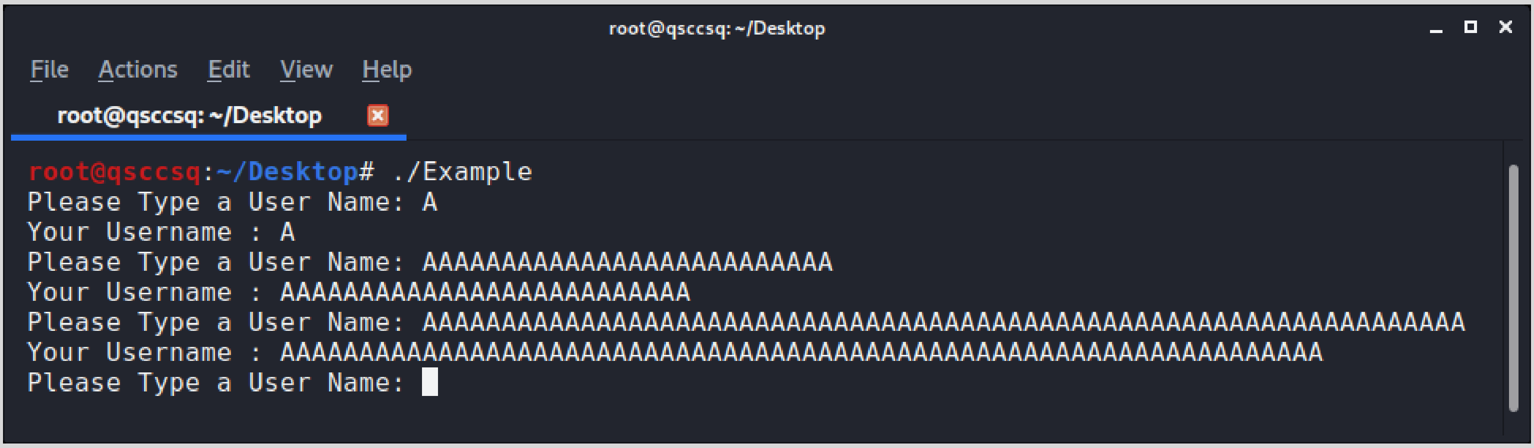
As we have mentioned many times before in our example, our main problem is that we exceed the data input limit given to us. There can be many solution stages of the said problem. In this example, we will prevent the target application from crashing using the GOTO statement. I will use the GOTO statement before the program closes;

```
Terminal-
File  Edit  View  Terminal  Tabs  Help
// Ethical Hacking - Sam Houston State University
#include <stdio.h>
int main ()
{

        char user_name[20];
        JUMP:
        printf ("Please Type a User Name: ");
        scanf ("%s",user_name);
        printf("Your Username : %s\n", user_name);
        goto JUMP;
        return(0);


}
                                                  1,49          All
```

Let's compile it;

```
root@qsccsq: ~/Desktop
File  Actions  Edit  View  Help
root@qsccsq: ~/Desktop
root@qsccsq:~/Desktop# gcc -no-pie -fno-stack-protector -z execstack Example.c -o Example
root@qsccsq:~/Desktop#
```

Let's run it;



Possible Question

1) Create Exam.c file;

   Codes:

```
#include <stdio.h>
#include <unistd.h>

int checker() {
    char getter[255];
    int x;
    register int id_getter asm("rsp");
    printf("Welcome to the Example of Buffer Overflow !\n");
    printf("\nEnter Your SHSU ID :\n", id_getter);
    x = read(0, getter, 510);
    printf("Your ID is : %s\n", getter);
    return 0;
}

int main(int argc, char *argv[]) {
    checker();
    printf("Thanks a Lot !\n");
    return 0;
}
```

2) Compile the Example.c file and put screenshots.
3) Make a fuzzer with Python then try to get segmentation fault with created application and put screenshots.

   (You can pass step 3 because it could be hard but there is an extra point right here. If you are not able to make a fuzzer try to put 272 times 'A')
4) Investigate Example.c with GDB as in document and put screenshots.
5) Try to debug malicious code block – each solution extra 10 points.

   (Hint: Try to set a limit, and loops.)