<u>Buffer Overflow Exploitation</u>

The purpose of this assignment is to help you learn to navigate GDB, and in the midst also learn about how programs are represented in machine language, how stack frames are structured in memory, and how programs can be vulnerable to buffer overflow attacks.

A buffer overflow occurs when the volume of data exceeds the storage capacity of the memory buffer. As a result, the program attempting to write the data to the buffer overwrites adjacent memory locations. Buffer overflows can affect all types of software. They typically result from malformed inputs or failure to allocate enough space for the buffer. If the transaction overwrites executable code, it can cause the program to behave unpredictably and generate incorrect results, memory access errors, or crashes.

Attackers exploit buffer overflow issues by overwriting the memory of an application. This changes the execution path of the program, triggering a response that damages files or exposes private information. If attackers know the memory layout of a program, they can intentionally feed input that the buffer cannot store, and overwrite areas that hold executable code, replacing it with their own code.

<u>Goal:</u>

Manipulate the program using GDB such that the function `secret()` is executed. This can be done by successfully accomplishing a buffer overflow exploit by taking advantage of a vulnerable line of code.

<u>Task 1:</u>

In this assignment, we have provided you with a vulnerable program “vuln.c”. You will use GDB to find the line of code that is vulnerable to a buffer overflow attack.

You will then exploit the buffer overflow vulnerability to reveal a secret message that is hidden in your program.

Compile the “vuln.c” file using this following command:

```
gcc -g -o vuln vuln.c -fno-stack-protector -z execstack -fstack-protector-all
```

Flags associated with the commands:

-g:

Produce debugging information in the operating system’s native format.

-z execstack:

mark the stack as executable.

-fno-stack-protector:

disable the stack protector feature, which checks for stack-based buffer overflows by inserting canaries into the stack.

-fstack-protector-all:

enable the stack protector feature for all functions, not just those that have buffer overflow detection annotations.

Task 2:

Use GDB on the binary:

`gdb vuln`

Go over the entire execution of program using related GDB commands. A few useful commands may involve:

break	set a breakpoint at the start of the specified function/line number/*memory address
run arg1	start the program with the specified command line arguments
Set {type} address value	set the value of a memory address to a specified value, where type is the data type (e.g., int, char, etc.)
continue	continue execution from the current breakpoint
next	execute the next line of code
step	execute the next line of code, stepping into function calls
print	print the value of the specified variable
info registers	show the current values of all registers
x/f (f = formate specifier.)	examine the memory at the specified address, where f is the format (x for hex, d for decimal, s for string, etc.)
disas	assembly dump

Task 3:

Identify the vulnerable line of code. **Write the vulnerability point and how you found this in your report. Explain why this piece of code is vulnerable**. **How could you protect against this?** You know that you've overflown the program when the program segfaults.

Task 4:

Once you get the segfault, you know you've overflown the frame pointer and that the next address should be the return address, where you want to place your malicious data. In this case, here it should be the starting address of the `secret` function. Use your GDB tools above to determine this location.

By writing to this address, we are corrupting the value that will be popped into the program counter when the leave instruction is executed, causing the program to jump to a different address than it should

You can inspect the stack pointer by using the following command:

`x/4x $sp`

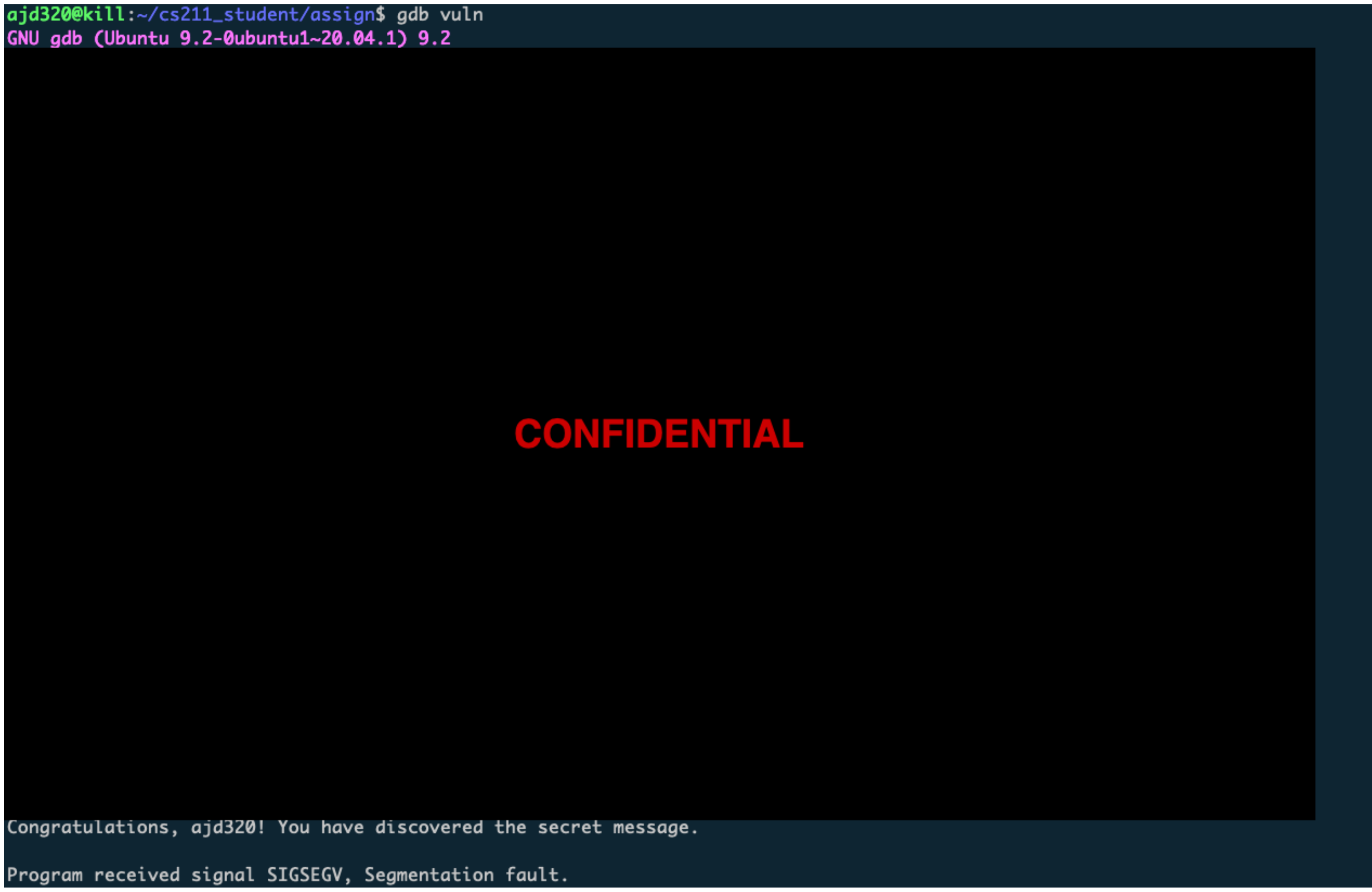
This will display the contents of the stack at the current stack pointer ($sp). The return address

Task 5:

Once you have the starting address of the secret function, you will want to overwrite the current return address with the starting address of secret. In turn, this will make the stack pointer now point at secret instead of going back to the main function. If done successfully, you will see the secret function print the following.

"Congratulations, {netID}! You have discovered the secret message."

Take a screenshot of your entire GDB session from the command `gdb vuln` all the way until the message "Congratulations, {netID}! You have discovered the secret message."



```
ajd320@kill:~/cs211_student/assign$ gdb vuln
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.1) 9.2

                                CONFIDENTIAL

Congratulations, ajd320! You have discovered the secret message.

Program received signal SIGSEGV, Segmentation fault.
```

<u>Note:</u>

1. You should scp the file to the iLab.
2. You should connect to iLab using ssh.
3. You should run and debug this program on the iLab.
4. Your report should include all necessary screenshots which clearly shows that you are running the program on iLab from your user account. It should also include all the steps it took to achieve the exploit.

<u>Submission instructions:</u>

1. Make your report in the word processor of your choice.
2. **Take a screenshot of your entire GDB session from the command `gdb vuln` all the way until the message "Congratulations, {netID}! You have discovered the secret message."**
    a. The screenshot should show all of your steps to successfully complete the buffer overflow as well as the resulting output.
3. Attach this screenshot as the first item in your report

4. Insert your report from Task 3.
5. Convert the file to the PDF format.
6. **Name your file as netid_report.pdf**
7. Submit to Canvas 1 file:
    a. netid_report.pdf