Look at main.c. Describe what it does. (2 points)

Task 1 - Compile main.c

Compile the program using the following command:

gcc -o main main.c -g

Type in or copy/paste this command

Notice that you get a warning.

Show a screen shot with your name somewhere in the screenshot, the compile command, and the warning. (2 points)

Task 2 – Run main from the debugger, without causing buffer overflow

Use the following command to enter the debugger

gdb main

Set breakpoints with the following commands

b get_input
b benign_function

Run the following command

disassemble main

Note that in the first line of the disassembly code that the stp function is run. stp is store pair and places two registers at the memory location given. x29 and x30 are placed on the stack. What are x29 and x30? Why are they placed on the stack? (2 points)

Run the program with the following command

r

You are now at the first line of benign_function.

Display hexadecimal register values with the following command.

i r

(hit enter to see the rest of the registers)

Fill in the following table with register values (hexadecimal not decimal) (5 points)

Register	Value
X29	
X30	
SP	

Table 1

Type n and return four times, so that you see "40 return z;"

We are now in the function benign_function.

We have two stack frames on the stack right now, one for main, and one for benign_function. The values of x29 and sp in table 1 above are the boundaries of benign_function's stack frame.

The stack frame for main should contain the values of x29 and x30 when the program first started. (Recall that we saw in the disassembly where these values got written to the stack.)

Look at the contents of the stack with the following command

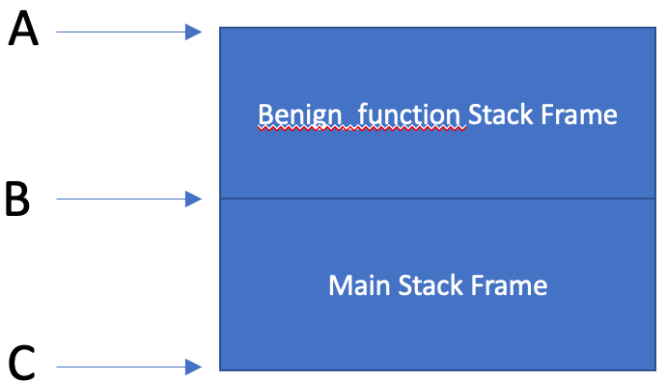
x/6gx $sp

Note that the TOP of the stack is in the TOP row of the table. So benign_function stack frame is at the top, then main stack frame.

You can figure out the boundaries of each stack frame using the current values of stack pointer and frame pointer, and by noticing the frame pointer value placed on the stack at the beginning of main.

Each row in the following table represents 16 bytes starting at the address shown. Copy the values for the bytes into Table 2. (5 points) Each column with a value is 8 bytes. A register is 8 bytes.
Table 2

Memory Address	Value	Value



Note: this drawing is meant to match the memory printout you did in an earlier step.

A is a LOWER address in memory than B and B is a LOWER address than C. The stack grows TOWARD lower addresses, so in this drawing the stack grows UP

Figure 2

Figure 2 shows a schematic of the stack frames. Using the values you entered into Table 1, what are the addresses pointed to by A and B in Figure 2? (2 points)

Given this information, use the values you entered into Table 2 to determine the previous values of x29 and x30 when they were written to the stack frame early in the execution of main? (5 points)

Using this information, what is the address pointed to by C in Figure 2? (3 points)
Type ***n*** repeatedly until you have entered get_input, main.c:46

Type

i r

Fill in the following table with hexadecimal register values (5 points)

Register	Value
x29	
x30	
SP	

Table 3

Type ***n*** repeatedly until the program is waiting for input (cursor is all the way on the left)

Type in the first 8 characters of your name and hit enter.

Now type ***x/6gx $sp***

Fill in the following table. (5 points)

Memory Address	Value	Value

Table 4

What does the data in memory shown in the top row represent? (What registers have been saved to memory?) (5 points)
Values of x29 and x30 when get_input was called

Circle or highlight in Table 4 the input you typed. (For reference, A will show up as 41, B as 42, and so on). (Also the input is backwards… yay little endian) (5 points)

Type ***n*** enough times to finish execution.
Note that the program exited normally.

Task 3 – Run main.c from the debugger, causing buffer overflow

Type ***r*** to run the program again

Type ***n*** repeatedly until the program is waiting for input.

Input a string that is 18 characters and press enter.

Type ***n***

Type

x/6gx $sp

Fill in the following table (5 points)

Memory Address	Value	Value

Table 5

What has happened to our stack frame for main? (5 points)

Type ***n*** repeatedly until you get some question marks

Display register values with the following command

i r

Fill in the following table with register values (5 points)

Register	Value
x29	
x30	
SP	

Table 6

What happened to our x29 and x30 value? (5 points)

Type ***n***

What is the result? Why did this happen? (4 points)

Type ***q***

Type ***y***

Task 4 – record address for arbitrary_code function

Now we are going to record the starting address for the arbitrary_code function. We are going to craft a buffer overflow attack that causes the x30 value that main saved to the stack to be overwritten with this address.

Type ***gdb main***

Type ***disassemble arbitrary_code***

What is the address in memory of the first line of the arbitrary_code function? (2 points)

Type ***q*** to exit the debugger

Task 5 - Run main from command line, overflow the buffer

Now we're back at the command line.

Run the main executable.

./main

Provide input that is 20 characters. What happens? (2 points)

Provide a screen shot with your name somewhere in the screen shot, the command to run main, and the result.

Task 6 - Run main from command line, craft input to main to execute arbitrary_code function

We're going to have the program jump to our arbitrary code function.

Type ***echo 0 > /proc/sys/kernel/randomize_va_space***

This command turns off ASLR. Research this and describe what it is and why we need to do this to make arbitrary_code function run. (8 points)

Now we will put an address into the buffer. We need to send hex characters. Do this with a command like this:

echo -e "ABCDEFGHIJKLMNOP\x78\xa8\xaa\xaa\xaa\xaa" | ./main

This pipes input to your program. The \x indicates a hex byte is coming. Craft your input so it goes to the address for arbitrary_code that you recorded in Task 3. Since we are doing little endian, the address has to be backwards in bytes.

What happens? (2 points)

(Control C to make execution stop)

Task 7 – Turn ASLR back on

Type ***echo 1 > /proc/sys/kernel/randomize_va_space***

Type ***echo -e "ABCDEFGHIJKLMNOP\x78\xa8\xaa\xaa\xaa\xaa" | ./main***

What happens? (2 points)

Task 8 – Fix the vulnerability

Write a new version of main.c, where gets(buffer) is replaced with fgets(buffer, 8, stdin)

Run your new main function with an input greater than 8 characters. What happens? (2 points)

Task 9

Go to https://cve.mitre.org

Find a stack buffer overflow vulnerability. Give the CVE number, the name of the vulnerability and the affected software. Look at the reference. How was it reported (e.g. GitHub, SourceForge)? Briefly describe the vulnerability (one sentence). (10 points) Why do you think there are still vulnerabilities like this? (2 points)