COEN225 Lab 2: Memory Corruption and Intro to GDB

Overview

The learning objective of this lab is for students to explore the contents of registers and memory in a running program using a debugger. Students will understand the layout of select variables allocated in memory and how unbounded memory writes can corrupt memory allocated for other variables.

Lab Submission

Perform the tasks below and submit your responses to questions noted in the task submission sections. Also include any source code you used to generate exploits or other tests.

The lab work and submission must be completed individually but working with other students is allowed. The Camino message board is the preferred method for asking questions so that the instructor and students can respond and allow all students to learn from it.

Lab Tasks

Task 1: Exploring a program

Task 1, Part 1 - The program

Download lab2a.c to your lab VM. Before you run anything, take a minute to read and understand the code.

Check your understanding: Review the code in lab2a.c. What does it do?

Compile lab2a.c using GCC. The -g3 flag is used here to enable debugging symbols which makes the binary easier to analyze in a debugger. Then run the program.

```
$ gcc -g3 lab2a.c -o lab2a
$ ./lab2a
```

Note: You may see some warnings when compiling lab programs using GCC, particularly with the use of the `gets()` function. More recent versions of compilers support newer standards that have deprecated some functions for security reasons (and we will see why). *Warnings* can be safely ignored for these labs, but you should not get any compiler *errors*.

The program will ask for user input. Try some input strings to understand how the program behaves before proceeding.

Check your understanding: Look at the code again. The variable `name` is a character array that holds your input string. How many bytes can this array hold?

Check your understanding: A string uses more data that the sum of its characters due to extra overhead of managing data of unfixed size. How many characters can `name` safely hold?

Think about the questions above before proceeding.

The answer to the above questions is 16 and 15. The array can hold 15 characters safely because the last byte must be a terminating null byte.

To test this, run the program 3 more times. Use an input of with a length of 15 characters, 16 characters, and finally 17 characters.

What was the output of each of the 3 inputs? If done correctly, `unused_string` is corrupted when the input is longer than the safe size of `name`, even though `unused_string` is *never modified in the source code*.

Let's dive deeper into what is happening using a debugger.

Task 1, Part 2 - GDB Navigation and Breakpoints
Load your lab program using GDB.

```
$ gdb ./lab2a
```

You should now be in the GDB command screen, denoted by `(gdb)` prompt. Use the run command to run the program.

```
(gdb) run
```

The program should behave normally as it does on the command line, and terminate after you enter a safe input. Now run the program again, but this time when you are at the input prompt, do not press Enter. Instead, press Ctrl+C, that is, holding down the control key and pressing C. This generates an interrupt signal that is caught by the debugger and puts you back into the command screen. It will also give you the memory address of the instruction that was running when the program was interrupted. Use the continue command to continue running the program.

```
(gdb) continue
```

You can interrupt a program any time you want using Ctrl+C, but this does not give you much precision it you want to pause a program at a specific point. Using a breakpoint allows you to specify a condition when a program should be interrupted. Try setting a breakpoint at the beginning of the main function and run the program.

```
(gdb) break main
(gdb) run
```

The program will automatically be interrupted at the start of the main function.

You can view all of the breakpoints that exist using the info command.

```
(gdb) info breakpoints
```

Here you can also see the memory address where the breakpoint is set. In our case, the debugger knew of the “main” symbol so that we could simply reference it in our command, but symbols are not always available. In cases where symbols are unavailable, breakpoints can also be set using a memory address.

```
(gdb) break *0x8048330
```

Breakpoints can also be set based on a specific line in the source code, if the debugger is aware of it. Use the list command to see the source code.

```
(gdb) list
```

Then set a breakpoint using the line number of the source.

```
(gdb) break 5
```

Look at your breakpoints now. You should have 3 if you have been following the instructions above. Breakpoints can be disabled the disable command followed by the breakpoint number identified by the "Num" column. Disabled breakpoints will show "n" under the "Enb" (Enabled) column.

```
(gdb) info breakpoints
(gdb) disable 1
(gdb) disable 2
(gdb) disable 3
(gdb) info breakpoints
```

Lastly, a conditional breakpoint can be created to only trigger if a given condition is true. Let's create a breakpoint that triggers only when the input is the string "ABC". Run the program and try different inputs to see when the breakpoint triggers.

```
(gdb) break 12 if strcmp(name, "ABC") == 0
(gdb) info breakpoints
(gdb) run
```

Creating the right breakpoints can significantly improve debugging efficiency. Let's reset our debugging environment by using the quit command to leave the debugger. Note that this will delete all of your breakpoints.

```
(gdb) quit
```

Task 1, Part 3 - Examining Memory

Launch the GDB debugger, set a breakpoint at the main function, and run.

```
$ gdb ./lab2a
(gdb) break main
(gdb) run
```

The breakpoint will place you before the execution of the first statement in the main function. Let's examine the current state of the program. Use the x command (for examine) to view memory contents. Examine the `argc` variable.

```
(gdb) x argc
0x1: <error: Cannot access memory at address 0x1>
```

You likely got an error message like the above. What is going on?

The `x` command treats the value provided as a memory address, and displays the data in memory at that address. In this case, the value of `argc` is 1 because no command line arguments were passed when the program was run. The output of the command is saying that it's trying to show the contents of memory address `0x1` but that memory address is invalid or otherwise inaccessible.

This isn't very useful to us, so let's examine the memory address of `argc`.

```
(gdb) x &argc
0xbffff410:     0x00000001
```

Like C, you can use the `&` operator to get a reference to a variable. The example output above tells us that the `argc` variable is stored at the address `0xbffff410`, and the integer value stored at that address is `0x1`.

Next, examine the `unused_string` variable.

```
(gdb) x unused_string
0xbffff3e0:     0x00000001
```

The unused_string variable is a c-string, a pointer (address) to the start of a contiguous block of memory containing characters. Since the value of the variable is already an address, examining it directly gives us the contents of the string. But since we stopped the program at the beginning of main, we haven't given it a useful value yet. Use the `next` command twice to execute the next 2 statements, then the `frame` command to check where you are in execution.

```
(gdb) next
(gdb) next
(gdb) frame
```

You should now be at the `printf()` statement. As this is a function, you may want to step into the function rather than moving to the next statement. For this, use the `step` command.

```
(gdb) step
(gdb) frame
```

You are now in the printf() function, but as this is a library function, you won't have debugging symbols for it (most Linux distributions have symbols as additional packages). Use the `finish` command to continue execution to the end of the current function.

```
    (gdb) finish
    (gdb) frame
    (gdb) next
```

You are now back in the `main()` function. Use the `next`, `step`, and `finish` commands to control the execution flow of the program within your debugging session. Let's go back to examining our variables.

```
    (gdb) x unused_string
```

Now that the program has executed a couple statements, the variable now 2has a different value, but it looks unfamiliar. This is because the data is not being formatted correctly for a string. In memory, all data is a sequence of numbers, even for characters. What you are seeing is the hexadecimal representation of the native word width (4 bytes) at the memory address of unused_string. To format the data as a string, tell the examine command to use the "s" display format specifier.

```
    (gdb) x/s unused_string
```

Much better. In general, you should always use a format specifier when using the examine command. To examine as hexadecimal, use the "x" display format.

```
    (gdb) x/x unused_string
```

As noted earlier, the hexadecimal display by default uses the native word width of 4 bytes. Using the string specifier ignores this as it prints characters until a terminating null byte is found. You can change the unit size (width) of the data read using the unit size specifier. You can use `b` for a byte, `h` for half word (2 bytes), `w` for word (4 bytes), or `g` for giant/double word (8 bytes). There is a special specifier `c` which displays the ASCII character representation of a byte.

```
    (gdb) x/c unused_string
    (gdb) x/xb unused_string
    (gdb) x/xw unused_string
    (gdb) x/xb &argc
    (gdb) x/xw &argc
```

You can specify how many units of each size should be displayed by including a number before the format specifier. Try some of these:

```
    (gdb) x/4c unused_string
    (gdb) x/4xb unused_string
    (gdb) x/s unused_string+4
```

In the last example above, we offset 4 bytes from the unused_string variable. Remember that string variables are simply addresses, so you can add or subtract from them.

In addition to variables, CPU registers also regularly hold addresses such as $esp (top of the stack) and $ebp (base pointer). Use the info command to see all registers. This will be useful in future labs.

```
(gdb) info reg
(gdb) x/8xw $esp
```

Check your understanding:

- What is the memory address of name?
- What is the memory address of unused_string?
- What variable's memory address is higher?
- How far away are they? Is there any space in between them?

Task 1 Submission

Explain the behavior observed earlier when entering inputs greater than 15 characters. Draw a memory diagram that captures the state of memory at the end of the `main` function. The diagram should include `name` and `unused_string`. Make note of which direction is low vs high memory, and the width should be 4 bytes as in the partial example below:

...				Low memory
'A'(name)	'B'	'C'	'D'(name+3)	
...				High memory

Task 2: Challenge

Compile and run lab2b.c. This program accepts user input and has a memory corruption bug similar to lab2a.c. The challenge is to corrupt the memory in a way that makes the program print "Success!".

```
$ gcc -g3 lab2b.c -o lab2b
$ ./lab2b
```

Tip 1: You must overwrite a variable with a specific set of bytes which can be found in the `check()` function. Convert the bytes to ASCII for easier input.

Tip 2: Use a debugger and set a breakpoint in the check function to examine the values being compared.

Tip 3: x86 is a little-endian architecture, which affects how multi-byte memory is written and read. A C-string is read/written as individual bytes one at a time. Integers (4 bytes) are read/written in a single instruction. The examine command in GDB behaves in the same way when using a multi-byte format specifier like w (4 bytes) and h (2 bytes).

Task 2 Submission

What was your input that got the success message? How does it work?

The way you entered the input to match the success code may feel "backwards". Why is this the case? (This is one of many reasons why exploits tend to be architecture and platform dependent)

END OF LAB