Overrun of intended bounds in a C program

1 Overview

This exercise illustrates overrunning the intended bounds of data structures in a C program.

1.1 Background

This exercise assumes the student has some basic C language programming experience and is familiar with simple data structures. No coding is required in this lab, but it will help if the student can understand a simple C program.

The GDB program is used to explore the executing program, including viewing a bit of its disassembly. However no assembly language background is necessary to perform the lab.

2 Lab Environment

This lab runs in the Labtainer framework, available at http://nps.edu/web/c3o/labtainers. That site includes links to a pre-built virtual machine that has Labtainers installed, however Labtainers can be run on any Linux host that supports Docker containers or on Docker Desktop on PCs and Macs.

From your labtainer-student directory start the lab using:

```
labtainer overrun
```

A link to this lab manual will be displayed.

3 Tasks

3.1 Review the mystuff.c program

A terminal opens when you start the lab. At that terminal, view the mystuff.c program. Use either `vi` or `nano`, or just type `less mystuff.c`.

3.1.1 The myData structure

Look at the `myData` structure. In the program, we declare the variable `my_data` to be a `myData` struct. Note the `public_info` character array has 20 elements. As with any array, we can refer to elements of the array using an index. For example, `my_data.public_info[4]` refers to the fifth character in the array, and `my_data.public_info[19]` refers to the very last character in the array.

If 19 is the very last character in the array, what would `my_data.public_info[20]` refer to?

3.1.2 Addresses of fields

After the program initializes the `my_data` structure, it displays the addresses of the start of the `public_data` field, and the `pin` field. And it displays the memory values of those fields.

3.1.3 Memory content

The program then enters a loop in which it allows the user to display hex values of individual characters within the `public_info` field. It is this loop that will let us explore the question asked earlier, nameley: what would `my_data.public_info[20]` refer to?

3.2 Compile and run the program

Use this command to compile the program:

```
gcc -m32 -g -o mystuff mystuff.c
```

Note the `-m32` switch creates a 32-bit binary and the `-g` switch includes symbols in the binary that will let us explore the program's execution using `gdb`.

Run the program:

```
./mystuff
```

and explore the values displayed at different offsets within (and beyond) the `public_info` field. Note the displayed address of the `public_info` field and the address of the `pin` field. How many bytes separate the two fields? Use the program to display the value of the pin field. Note that if your `fav_color` buffer size is odd, the compiler will *pad* the buffer so that the next variable starts on 4-byte word boundary.

3.3 Explore with gdb

Run the program under the GDB debugger:

```
gdb mystuff
```

Use the `list` command to view the source code. Set a breakpoint in the `showMemory` function on the line where it will print the value at the given offset. (Use `list showMemory` to view source for that function.) And then run the program from within gdb:

```
break <line number>
run
```

When the program hits the breakpoint, display 10 words (40 bytes) of system memory as hex values starting at the `data` structure:

```
x/10x &data
```

Does the memory content correspond to what you observed while running the program?

3.3.1 Extra exploration

Set a breakpoint at the end of the `handleMyStuff` funtion, i.e., on the line of the final right brace (}) in that function. Then continue with the `c` command. At the prompt for the next offset, enter a q. Then, when the program hits the breakpoint, display the disassembled program using:

```
display/i $pc
stepi
```

And single step through the remainder of the `handleMyStuff` function disassembly by repeatedly pressing the `Return` key until the program gets to the `ret` instruction.

This is the point in the program at which the `handleMyStuff` function will return to the `main` function. The `ret` instruction directs the processor to jump to the instruction at the address contained at the current stack pointer. Display the memory content pointed to by the stack register using:

```
x $esp
```

The displayed value will become the next instruction address, which you can confirm using one more `nexti`. Make note of that current instruction pointer. Look again at the stack address that held this return value. Note that it is higher than the address of the `data` structure observed in the `showMemory` function. Compute and record the difference between the two addresses.

Rerun the program outside of the debugger and use it to display the return address value, one byte at a time. Confirm that address is what you observed in gdb[1]. Imagine that the program let us modify the individual items in the `public_info` array. When the program hits the `ret` instruction that you viewed in gdb, it would then *return* to an address you wrote.

4 Submission

After finishing the lab, go to the terminal on your Linux system that was used to start the lab and type:

```
stoplab
```

When you stop the lab, the system will display a path to the zipped lab results on your Linux system. Provide that file to your instructor, e.g., via the Sakai site.

This lab was developed for the Labtainers framework by the Naval Postgraduate School, Center for Cybersecurity and Cyber Operations under sponsorship from the National Science Foundation.

[1]The address may appear backwards to you. Don't let that hang you up, it is an artifact of machine architecture that you can learn about by googling *Endianess*