Introduction to printf memory refererences

1 Overview

This exercise introduces the printf function and encourages the student to explore the manner in which the function references memory addresses in response to its given format specification. This lab provides an introduction to techniques that are used in the more advanced printf labs (formatstring and format64).

1.1 Background

This exercise assumes the student has some basic C language programming experience and is somewhat familiar with the use of gdb[1]

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. Some assembly language background would be helpful in performing the lab, but is not necessary.

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 printf
```

A link to this lab manual will be displayed.

3 Tasks

3.1 Review the printTest.c program

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

Observe the syntax of the first `printf` statement. The first parameter is a format string that contains literal text to be displayed, and one or more one or more *conversion specifications* that determine how any remaining parameters are displayed. The conversion specification begins with the `%` symbol. In the first printf statement, the conversion specification is a `%d`, which directs printf to display the parameter as an integer. Thus, the value of `var1` would be displayed as an integer following the string ”`var1 is:  `”. The `\n` “escape n” sequence causes printf to generate a newline.

The second printf statement illustrates how we can display the values of multiple parameters. In this case, the hexadecimal representation of an integer (the `%x`) followed by a string (using the `%s` conversion specification).

The printf function has an extremely rich set of conversion specifications, but most those are not important for this lab. What **is** important for this lab is the manner in which printf references memory to find the values to be displayed.

[1]This lab manual provides detailed gdb commands to accomplished the prescribed tasks, and can serve as an introduction to gdb.

The third printf statement is vulnerable to mischief, as we will see in this lab.

3.2 Run printTest

The `mkit.sh` script will compile the program as a 32-bit executable:

```
./mkit
```

You may then run the program:

```
./printTest
```

and observce its output.

3.3 x86 function calling conventions

When a 32-bit x86 program is about to call a function, the parameters to the function are first pushed onto the stack. The function is called and the function references its parameters from the stack. In the first printf, there are two parameters: the format string and the `var1` variable.

Since in the 32-bit x86 the stack pointer register `esp` decreases as the stack ”grows“, the figure 1 diagram has low memory at the top of the diagram.

```
low memory

        [stuff used by printf]

esp ->  pointer to the format string
        var1 value

        [stuff from calling function]
high memory

       Figure 1: Stack prior to call to printf
```

In figure 1, we see the var1 value has been pushed on the stack, followed by the pointer of the format string.

3.4 Behavior of printf

When the printf function is called, it expects to find the pointer to the format string at the top of the parameters on the stack. It then reads the format string and interprets the conversion specifications. In the case of our first printf, it only sees the `%d`, which causes printf to treat the next parameter on the stack as an integer, and display its value as such along with the rest of the format string literals.

The second printf function call will have three parameters. This time, the printf function sees a `%x` conversion specification and looks at the next parameter, which is now the `var2` value and it displays that as a hexadecimal value per the `%x`. It then sees the `%s` and treats the next parameter as a pointer to some string, which it then displays.

3.5 Observe calling conventions with gdb

Run the program in gdb:

```
gdb printTest
```

List the program with the `list` command at set a breakpoint at the line of the first printf statement and run:

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

The program will break just before the call to printf. But not close enough for our purposes, so we will view the disassembly of the machine instructions so that we can advance execution to just before the actual call. Use this gdb command to display the disassembly of the current instruction:

```
display/i $pc
```

Then use the `nexti` instruction to advance execution to the next instruction. Repeatedly press the Return key to keep stepping until you reach the call to `printf@plt` Now the program is really just about to call printf. Look at twenty words on the stack as hexidecimal values:

```
x/20xw $esp
```

The `esp` register is pointing to the top of the stack, which contains the first parameter to printf, i.e., the pointer to the format string. Confirm that by examining memory at that address (i.e., the first displayed word) as a string:[2]

```
x/s <address>
```

You should see the format string. The word at the next parameter on the stack is our `val1` value of 13 (hex 0x0d).

Look at the content of subsequent addresses. You see some address values and such, but a bit further in you will see the two values of `var1` and `var2` within adjacent words. That memory is where the printTest program has stored those two values. You previously observed a copy of the `var1` value near the top of the stack. The values at the higher addresses are the original values of those variables.

Our next step will be to fool printf into displaying those values from their original locations.

If you'd like to review what you've seen a bit more, set a breakpoint at the 2nd printf, step through its disassembly until that call, and look at the stack to identify the three parameters to printf.

3.6 User input in format strings

Look at the source code of the testPrint.c program again, and find the line that reads:

```
printf(user_input);
```

In this case, the format string is supplied by the user, and there are no other parameters to be displayed. What if the user supplies a format string that contains conversion specifications? The printf function has no way of knowing the providence of the format string, nor does it have any way of knowing the number of parameters provided in the function call – it simply assumes parameters have been pushed onto the stack. Thus, if printf encounters a `%x` in the format string, it will look at the next parameter on the stack, and since there were no other parameters, it will find whatever happened to be at that address. Lets expand our repertoire of conversion specifications to include:

[2]cut/paste by highlighting the desired text and pressing `ctl shift c` and then paste that with `ctl shift v`

```
%8x
```

which directs printf to display the word as an 8 digit hexadecimal value. We'll combine a raft of those format conversions and provide that as input when the program prompts us for a string

```
AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.
```

Run the program (without gdb) and provide the above string as input. Where do the displayed values come from? Run the program in gdb again, this time set a break at line number of the vulnerable call to printf and use `run` to start the program. Before the program reaches your breakpoint, it will primpt you to enter the string. Paste the above string and the program will then break at the (almost) call to printf. Use

```
display/i $pc
nexti
<return>....
```

to step to the call to `printf@plt` and then display the stack content.

```
x/20x2 $esp
```

Find the first (and only) parameter to the printf statement and confirm it is the address of your user-provided format string:

```
x/s <address>
```

The use the `c` command to continue, allowing the program to output the results of the printf statement. Compare that output to what you see in memory just past the address of the format string.[3]

3.7 More detail

See the formatstring lab to further explore printf vulnerabilities, including a method for modifying the content of memory.

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.

[3] You may notice the content of memory changes between each run of the program. This is due to Address Space Layout Randomization. Google it.