LABORATOIRE VIII – CEG 4399

CEG 4399- Design of Secure Computer Sys.

Université d'Ottawa

Professeur : Amir H. Razavi

Noms et numéros des étudiants :
Gbegbe Decaho Jacques 300094197

Date de soumission: 14 November 2024

Introduction to printf memory references

Introduction

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.

```
student@Labtainer-VirtualBox:~/labtainer/labtainer-student$ labtainer printf
latest: Pulling from labtainers/printf.printf.student
9b5b81050711: Pulling fs layer
b124fca817fe: Pulling fs layer
337c643669d0: Downloading [========>                                          ]     7
9b5b81050711: Downloading [=========================================>         ]     7
9b5b81050711: Downloading [==================================================>]     9
9b5b81050711: Extracting [==================================================>]     93
9b5b81050711: Extracting [==================================================>]     93
9b5b81050711: Pull complete
b124fca817fe: Extracting [==================================================>]     12
```

We loaded the "**printf**" labtainer into the virtualbox in order to launch our working environment.

3. Task

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`.

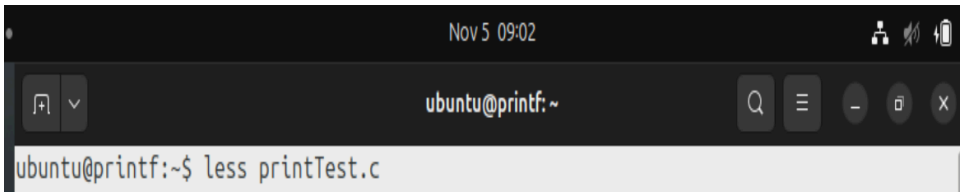
Observe the syntax of the first `printf` statement. The first parameter is a format string that contains literal text to be displayed, and one or more one or more *conversion specifications* that determine how any remaining parameters are displayed. The conversion specification begins with the `%` symbol. 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.

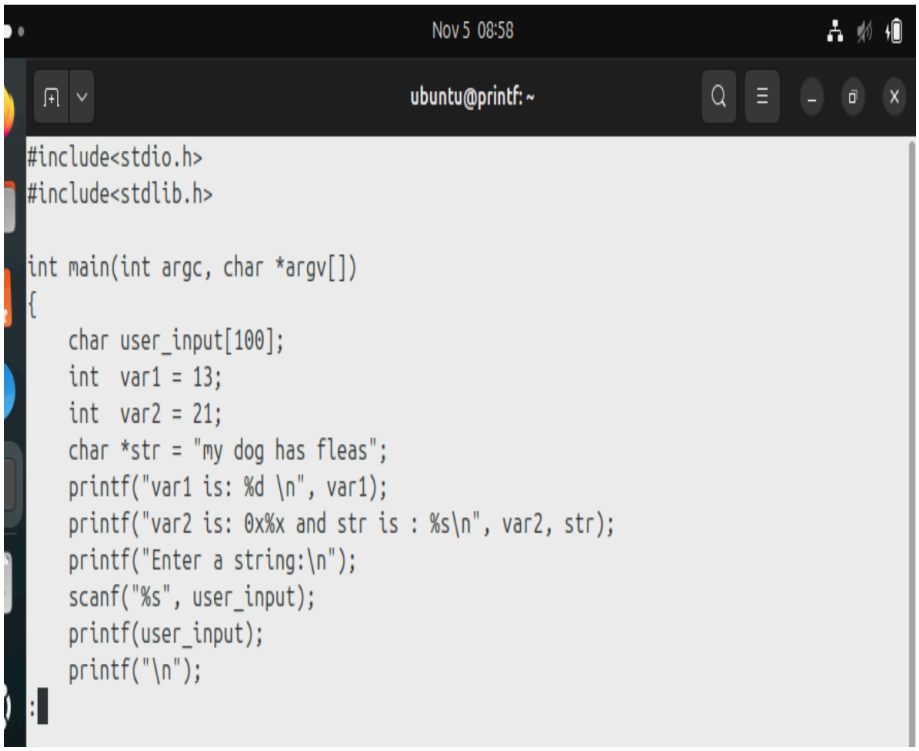
The second printf statement illustrates how we can display the values of multiple parameters. In this case, the hexadecimal representation of an integer (the `%x`) followed by a string (using the `%s` conversion specification).

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.

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

After running the labtainer, we were able to observe the windows below "**ubuntu@printf**"





With the command "**less printTest.c**" we were able to view the code contained in the printTest file.

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.

With the command "**./mkit**" we got a message that the file was not found.

```
ubuntu@printf:~$ ./mkit
-bash: ./mkit: No such file or directory
```

```
ubuntu@printf:~$ ./printTest
var1 is: 13
var2 is: 0x15 and str is : my dog has fleas
Enter a string:
decaho-1
decaho-1
decaho-1
```

```
ubuntu@printf:~$ gcc -m32 -o printTest printTest.c
printTest.c: In function 'main':
printTest.c:15:11: warning: format not a string literal and no format arguments [-Wfo
rmat-security]
   15 |    printf(user_input);
      |           ^~~~~~~~~~
```

In order to directly execute the file, we had the option to execute it directly with "**./printTest**" or with c commands like "**gcc -m32 -o printTest printTest.c**"
The output shown above is a security warning generated by the compiler because the line **printf(user_input)**; directly uses the string entered by the user as a format without validation. This makes the program vulnerable to "**format string**" attacks where a malicious user could execute unexpected code by manipulating the input.

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.

This section explains how the stack works
Before a function like printf is called on a **32-bit x86 architecture**, its parameters are pushed onto the stack. The stack grows to lower memory addresses, causing the **esp register** to shrink with each new addition. In the example **printf("var1 is: %d\n", var1)**;, the address of the format string is pushed first, followed by the value of **var1**. So, just before the call, the stack contains the address of the format string and the value of **var1**, with the data from the calling function higher in memory.

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.

Before "**printf**" is called on a **32-bit x86 architecture**, its parameters (such as the format string and variables) are pushed onto the stack, which grows to lower memory addresses. The "**$esp**" register points to the top of the stack and shrinks with each addition. In "**printf("var1 is: %d\n", var1);**", the address of the format string is pushed first, followed by the value of "**var1**". So, just before the call, the stack contains these items in that order, with the parameters of **"printf"** at the bottom, followed by the other data from the calling function higher up the stack.

3.5 Observe calling conventions with gdb

Run the program in gdb:

```
gdb printTest
```

In this part we launched the **gdb** program in the **printf** window we were able to observe that the program is functional above.

```
For help, type "help".
Type "apropos word" to search for commands related to "word
Reading symbols from printTest...
(gdb) list
1       #include<stdio.h>
2       #include<stdlib.h>
3
4       int main(int argc, char *argv[])
5       {
6           char user_input[100];
7           int  var1 = 13;
8           int  var2 = 21;
9           char *str = "my dog has fleas";
10          printf("var1 is: %d \n", var1);
```

With the "**list**" command we were able to observe the contents of the **printTest** file.

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
```

We added a **break** at line **10** shown below.

```
(gdb) break 10
Breakpoint 1 at 0x129e: file printTest.c, line 10.
```

```
(gdb) run
Starting program: /home/ubuntu/printTest

Breakpoint 1, main (argc=1, argv=0xffffd644) at printTest.c:10
10          printf("var1 is: %d \n", var1);
```

After using the "**run**" command, which showed us where the **breakpoint** was applied in the program.

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 with the command **"dis**play/i $pc", we were able to display the disassembly with its equivalence in the **form of a hexadecimal number** at the start of the line.

```
(gdb) nexti
0x565562a1      10          printf("var1 is: %d \n", var1);
1: x/i $pc
=> 0x565562a1 <main+84>:        pushl  -0x7c(%ebp)
2: x/i $pc
=> 0x565562a1 <main+84>:        pushl  -0x7c(%ebp)
(gdb)
0x565562a4      10          printf("var1 is: %d \n", var1);
1: x/i $pc
=> 0x565562a4 <main+87>:        lea    -0x1faf(%ebx),%eax
2: x/i $pc
=> 0x565562a4 <main+87>:        lea    -0x1faf(%ebx),%eax
(gdb)
0x565562aa      10          printf("var1 is: %d \n", var1);
1: x/i $pc
=> 0x565562aa <main+93>:        push   %eax
```

With the "**nexti**" command, we were able to advance in the execution up to the **printf call** shown above at the bottom of the image.

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
```

```
(gdb) x/20xw $esp
0xffffd4f0:     0x56557019      0x0000000d      0x000000c2      0x5655626b
0xffffd500:     0xffffd53a      0xf7ffc89c      0xf7ffc8a0      0xffffd644
0xffffd510:     0xf7ffd000      0xf7ffc8a0      0xffffd53a      0x0000000d
0xffffd520:     0x00000015      0x56557008      0x00000001      0xf7ffc7e0
0xffffd530:     0x00000000      0x00000000      0x00005034      0xb9576400
```

With the use of the command "**x/20xw $esp**" we were able **to display** the total contents of the stack.

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.

```
(gdb) x/s 0x56557019
0x56557019:     "var1 is: %d \n"
```

We then use the in-memory address to display the contents of printf which is: "**var1 is: %d \n**".

```
(gdb) quit
A debugging session is active.

        Inferior 1 [process 1240] will be killed.

Quit anyway? (y or n) y
```

We exited the **gdb** as requested in the instructions before continuing into the next part of the lab.

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:

```
%8x
```

```
(gdb) run
Starting program: /home/ubuntu/printTest
var1 is: 13
var2 is: 0x15 and str is : my dog has fleas
Enter a string:
 %8x
 %8x

Breakpoint 1, main (argc=1, argv=0xffffd644) at printTest.c:15
15          printf(user_input);
```

We then executed the c code **printTest** in a **gdb** and as test character we chose "**%8x**" then observed the result followed by the **breakpoint**.
Shown above.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/ubuntu/printTest
var1 is: 13
var2 is: 0x15 and str is : my dog has fleas
Enter a string:
AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x
AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x

Breakpoint 1, main (argc=1, argv=0xffffd644) at printTest.c:15
15          printf(user_input);
```

```
(gdb) display/i $pc
1: x/i $pc
=> 0x565562f3 <main+166>:       sub    $0xc,%esp
(gdb) nexti
0x565562f6      15          printf(user_input);
1: x/i $pc
=> 0x565562f6 <main+169>:       lea    -0x70(%ebp),%eax
(gdb)
0x565562f9      15          printf(user_input);
1: x/i $pc
=> 0x565562f9 <main+172>:       push   %eax
(gdb)
0x565562fa      15          printf(user_input);
1: x/i $pc
=> 0x565562fa <main+173>:       call   0x565560b0 <printf@plt>
```

We ran the program again in the gdb with a string as the test choice this time: "**AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.**".

We observe the doubled output illustrated in the screen above. After which we went through **the disassembly** to the memory address where the "**printf(user_input)**" was stored, identified by "**0x565560b0 <printf@plt**>".

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.%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
```

```
ubuntu@printf:~$ ./printTest
var1 is: 13
var2 is: 0x15 and str is : my dog has fleas
Enter a string:
AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x
AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x
AAAAffe78f48.58215008.5821426b.ffe78f5a.ebaf389c.ebaf38a0.ffe79064.ebaf4000.ebaf38a0.
ffe78f5a.       d.      15.58215008.41414141.2e783825.2e783825
```

We then re-run the "**printTest**" program but this time **outside of gdb**.
Then we put the last string back as the string choice.
We were then able to observe that the result is a **doubled output**, then a third line containing the memory addresses where the print and the **breakpoint** are stored.
Which after comparison turns out to be identical to those encountered in the display using **gdb**.

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]

In order to check if our hypothesis is proven we **launched the gdb** in order to recheck it.

```
(gdb) display/i $pc
1: x/i $pc
=> 0x565562f3 <main+166>:       sub    $0xc,%esp
(gdb) nexti
0x565562f6      14          printf(user_input);
1: x/i $pc
=> 0x565562f6 <main+169>:       lea    -0x70(%ebp),%eax
(gdb)
0x565562f9      14          printf(user_input);
1: x/i $pc
=> 0x565562f9 <main+172>:       push   %eax
(gdb)
0x565562fa      14          printf(user_input);
1: x/i $pc
=> 0x565562fa <main+173>:       call   0x565560b0 <printf@plt>
```

```
(gdb) x/20xw $esp
0xffffd4f0:    0xffffd528    0xffffd528    0x56557008    0x5655626b
0xffffd500:    0xffffd53a    0xf7ffc89c    0xf7ffc8a0    0xffffd644
0xffffd510:    0xf7ffd000    0xf7ffc8a0    0xffffd53a    0x0000000d
0xffffd520:    0x00000015    0x56557008    0x41414141    0x2e783825
0xffffd530:    0x2e783825    0x2e783825    0x2e783825    0x2e783825
(gdb) x/s 0xffffd528
0xffffd528:    "AAAA%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x.%8x"
```

We repeated the same steps as before. using the "**display/i $pc** and **nexti**" to advance to the call of "**printf**" which is "**printf@plt**".
Then, we observed the contents of **the stack** once more.
And finally, with the command "**x/s 0xffffd528**", we confirmed that the **first value** displayed is indeed the address of the format string.

Below, using the command "**c**", we were able to continue and compare the output of the program with what we observed in memory, just after the address of the format output.

```
(gdb) c
Continuing.
AAAAffffd528.56557008.5655626b.ffffd53a.f7ffc89c.f7ffc8a0.ffffd644.f7ffd000.f7ffc8a0.
ffffd53a.      d.      15.56557008.41414141.2e783825.2e783825
[Inferior 1 (process 388) exited normally]
```

```
(gdb) Quit
(gdb) ubuntu@printf:~$
```

We terminated the gdb program and then closed the "**ubuntu@printf**" window

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.

We finally stopped the labtainer with the "**stoplab**" command.