ENPM691 Homework Assignment 7

Hacking of C programs and Unix Binaries-Fall 2024 lcollier

Dipam Bhattacharyya, MEng Cybersecurity dipamb2@umd.edu

***Abstract*—The assignment aims to exploit format string vulnerabilities through the "printf" function. It will attempt to invoke a function that calls a child process.**

***Index Terms*—homework, hacking, C, coding**

I. Introduction

Format strings are strings that include format specifiers. These are utilized in format functions in C and various other programming languages. The format specifiers within a format string act as placeholders, which will be substituted with data provided by the developer.

The vulnerability arises when there is a discrepancy between the number of format specifiers in the string and the number of arguments supplied to fill those specifiers. If an attacker provides more placeholders than available parameters, they can exploit the format function to read from or write to the stack.

In this experiment, through crafted payloads, an attempt will be made to overwrite the return address of main to point to another function within the code. This function will then invoke a child program that executes code as root.

II. Methodology

A. Host environment and architecture

[1] The experiment was done on a Ubuntu v16.04 VM with Windows x64 as the host PC. All exploit protection settings in Windows are default, including ASLR(default ON). To check the VM architecture, the "lscpu" terminal command was given:

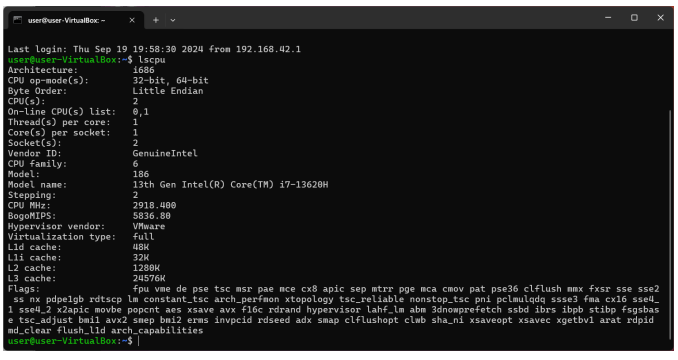

Fig. 1. Ubuntu CPU information

B. Format Strings in C

In C, format strings work as shown in Fig.2. For three "%d" signs, three arguments are passed in "printf()". In the stack, the top will have the format specifier followed by the three arguments in order of being printed. If we do not pass these arguments, and only "%d"'s, the "printf" will start reading values off the stack. Also, since here arguments are not passed, format specifiers will follow local variables of the function stack. The first "%d" will read the immediate value next to it(in the next higher address) followed by other values if more are defined.

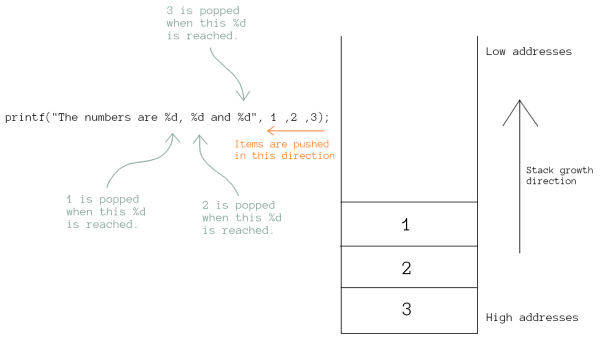

Fig. 2. Format specifiers in memory

C. "%x" and "%n" Formatter

The "%x" specifier reads the hexadecimal value on the memory address. If some width is given, like "%20x", it first prints 20 bytes in stdout followed by the hex value in memory.

On the other hand, "%n" expects an address as input and writes the count of the number of bytes printed in stdout to that address. It will store the hex equivalent value to the address referenced to it.

D. GDB debugging

[2] After the generation of the executable, we are attaching gdb to it. Then, we are looking at both source and disassembled code together while also having terminal access.

$$(gdb)\ layout\ split \tag{1}$$

To check the stack frames(in reverse order of being invoked) used in the program:

$$(gdb)\ bt \tag{2}$$

And to inspect a particular frame:

$$(gdb)\ info\ frame\ 1 \tag{3}$$

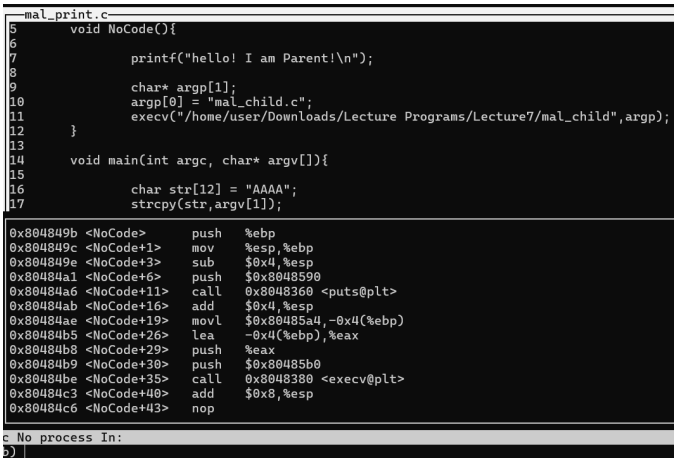

Fig. 3. gdb split layout output

To check the contents of memory from a particular point, say pointer or absolute address, the following(See Fig.4) is used. As an example, 20 words from the stack pointer can be seen:

$$(gdb)\ x/20x\ \$sp \tag{4}$$

The stack pointer is pointing to address "0xbffff588" with value "0x08048509". There are 4 columns and each entry is of 4 bytes (size of 1 word). In the first column, the remaining values seen are the contents of adjacent memory locations.

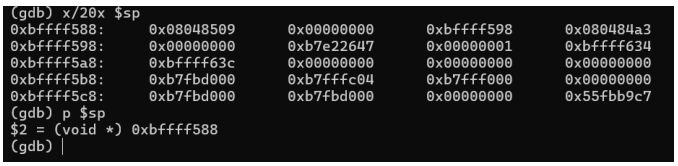

Fig. 4. 20 words displayed from stack pointer

E. Code Input Generation

Perl command line can be used conveniently to generate strings and feed the code. The advantage here is that it gives us the option to create hex strings using ASCII.

Fig. 5. String generation using Perl

The following method can be used as input in the GDB interface. It evaluates to hex characters:

$$(gdb)\ r\ \$(print\ -e\ 'hexstrings') \tag{5}$$

F. Code Analysis and Verification

For the experiment, the "mal_print.c" code (See Appendix A) is used. Another child program is written "mal_child.c" (See Appendix B) which will be called by the primary code. The "mal_print" takes two command line arguments and copies the first one to the buffer of 12 bytes. The buffer is initialized with a set of 4 byte "A"s. Then it prints the second argument on the stdout.

Another function is present "NoCode" which is not called by the "main()". This user function prints a message and then calls the child executable present in memory. The child code in turn prints a message and executes a system command as root.

Here, we are providing the address of the "main" return pointer as the first CLI input. This we found out by printing "$ebp+4", meaning the previous value in the stack of base pointer. The "strcpy()" copies the 4-byte address to the buffer variable. Next, we are printing 'a' bytes in stdout through "%ax" where 'a' is the decimal equivalent of the "NoCode()" function address in the code section of memory. Combined with this, "%1$n" is put which will refer to the value stored in the buffer address in the stack. So this is the second CLI input.

Essentially what these inputs are doing is that "%n" will convert the bytes of decimals printed (which translates to an actual address of "NoCode") and overwrite the return pointer of "main". The buffer variable has the address of the return pointer referenced by "%n".

III. Results

Figure 6 shows that "mal_child" got called by "mal_print.c" and it executed shell as root.

Figure 7 is the state of memory before the command line arguments are copied to the function stack. Color blue is the format specifiers, green is the buffer, yellow is the base pointer and red is the return address of the main function.

Figure 8 depicts how the value of the main return address pointer to lib got modified to the user function.

Figure 9 shows how to figure out the address of the return pointer by traversing the stack.

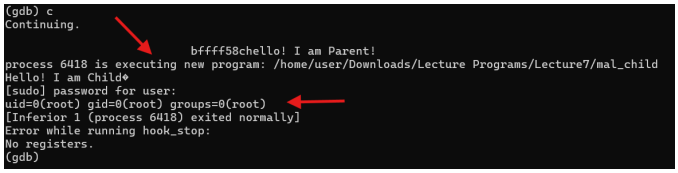

Fig. 6. Child program called and executed system commands as root

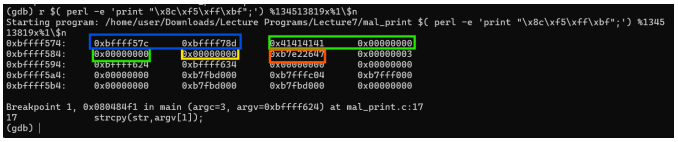

Fig. 7. Contents of the buffer before strcpy()

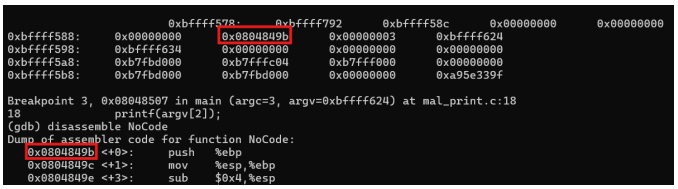

Fig. 8. Contents of the buffer after strcpy()

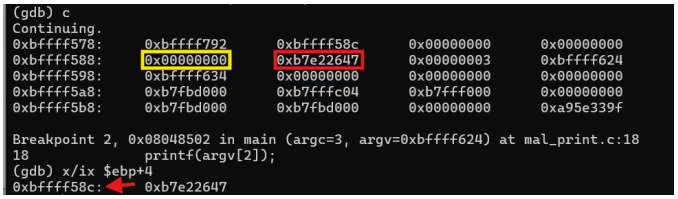

Fig. 9. Figuring out the address of the return pointer

IV. Discussions

A. Overwriting buffer variable with return address

As seen in Fig.7, a set of initial ”41” (green) is present as a buffer value. The red marked ”0xb7e22647” is the value of the internal library to be called after ”main()” finishes. When we input a hex string to GDB (See Appendix C, Fig.5) and run, the ”strcpy()” copies it to the buffer. Here, as Fig.9 shows, that string is ”0xbfff58c” which is where the return pointer lives. Notice how that address is present in the buffer instead of ”41” (value directly above red marked section) in Fig.9.

With the first CLI input, we successfully planted an address at local buffer space.

B. Invoking user function and child process

Second, we use ”134513819” as the decimal equivalent of the address where ”NoCode()” lives in ”%x” width. This prints 134513819 bytes in stdout. Following this, we add ”%1n” which converts ”134513819” to ”0x0804849b” and stores it at the address pointed by ”0xbfff58c”. It is possible because ”0xbfff58c” is next in stack as buffer variable value. Since ”0xbfff58c” is the address of the main return pointer, it is overwritten with ”0x0804849b” from the original ”0xb7e22647” (Seen in Fig.8).

Once the main execution finishes, it returns to ”NoCode()” instead of the C library, and our user function is called. This function using ”execv()” called the child executable. The executable ”mal_child” is run and prints the system command as root as seen in Fig.6. The figure clearly shows the printed sample ”hello” messages from both of the programs.

Finally, we got the child program to be invoked as an SU process through our second CLI input.

C. Unexpected Results

One of the failed attempts was while trying to overwrite the GOT entries. Instead of modifying the return pointer, a ”printf” or ”puts” entry in GOT could be used and point to the user function. Even though protections were off, it threw errors as non-writable addresses.

D. Limitations and Considerations

The experiment was done on an Ubuntu VM and these address mapping changes if performed on another OS, or if executed with ASLR turned on. If the code was compiled with a non-executable stack, the function won’t run. Additionally, the code and stack locations are static, so it was easier to add an absolute address as input. Moreover, different compilers may address memory differently than our GCC compiler.

Appendix A
The ”mal_print.c” code:

```
#include<stdio.h>
#include<string.h>
#include<unistd.h>

void NoCode(){

        printf("hello! I am Parent!\n");

        char* argp[1];
        argp[0] = "mal_child.c";

        /*calling the child process*/

        execv("/home/user/Downloads\
        /Lecture Programs/Lecture7/mal_child"\
        ,argp);
}

void main(int argc, char* argv[]){

        char str[12] = "AAAA";
        strcpy(str,argv[1]);
        printf(argv[2]);

return 0;
}
```

Appendix B
The ”mal_child.c” code:

```
#include<stdio.h>
```

```
int main(int argc, char* argv[]){

        printf("Hello! I am Child!\n");

        /*execute command as root*/
        system("sudo -i id");

return 0;
}
```

Appendix C
Run command in GDB with two ”argv[]” values

```
r $( perl -e 'print "\x8c\xf5\xff\xbf";')\
%134513819x%1\$n
```

REFERENCES

[1] Bhattacharyya, D.: In: ENPM691 Homework Assignment 3. pp. 1–3 (2024)

[2] Bhattacharyya, D.: In: ENPM691 Homework Assignment 2. pp. 1–4 (2024)