

2. 冷补丁 getshell

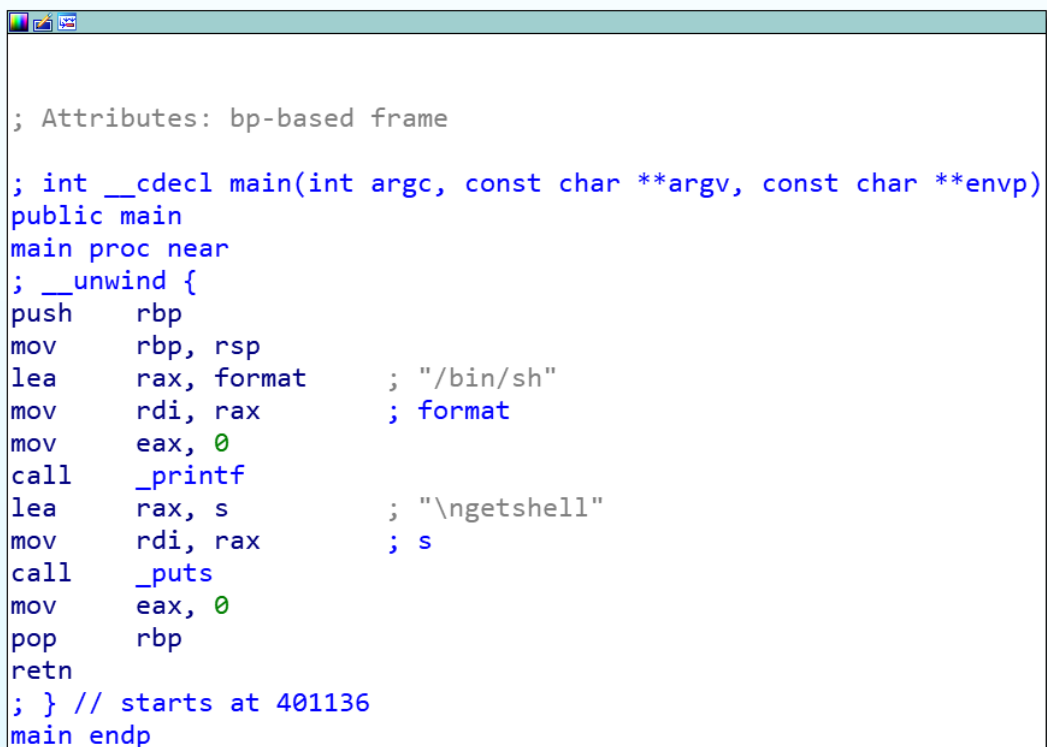
2.1 实验要求

- 程序 `getshell` 调用 `printf()` 函数打印了 `/bin/sh` 字符串，利用这一点可以实现 `getshell`。
- 要求利用 `LIEF` 库，将 `getshell` 程序中的 `printf()` 函数替换为补丁函数 `newprintf()`，获取系统的控制权，在当前目录下写入包含学号的文件，再将该文件显示出来。

2.2 实验过程

2.2.1 程序分析

- 首先使用 `IDA` 打开 `getshell` 程序，其 `main` 函数的反汇编结果如下：



```
; Attributes: bp-based frame

; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near
; __unwind {
push    rbp
mov     rbp, rsp
lea     rax, format      ; "/bin/sh"
mov     rdi, rax          ; format
mov     eax, 0
call    _printf
lea     rax, s            ; "\ngetshell"
mov     rdi, rax          ; s
call    _puts
mov     eax, 0
pop     rbp
retn
; } // starts at 401136
main endp
```

- 将 `main` 函数反编译，结果如下：

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     printf("/bin/sh", argv, envp);
4     puts("\ngetshell");
5     return 0;
6 }
```

2.2.2 补丁编写

- 由于要将 `printf()` 替换为补丁函数 `newprintf()` 以实现 `getshell`，且已有 `/bin/sh` 字符串可作为 `getshell` 时的可用参数（实际上，如果没有该字符串，也可以使用汇编语句在栈中手动构造该字符串），因此考虑在 `newprintf()` 中使用 `sys_execve` 系统调用，`x86_64` 架构下 `sys_execve` 系统调用的相关信息如下：

rax	System Call	rdi	rsi	rdx
59	sys_execve	const char* filename	const char* argv[]	const char* envp[]

- 据此构造 `newprintf()` 如下：

```
1 // hook_printf.c
2 void newprintf() {
3     asm (
4         "movq $0, %rsi\n"
5         "movq $0, %rdx\n"
6         "movq $59, %rax\n"
7         "syscall\n"
8     );
9 }
```

- 编译该补丁文件生成一个静态函数库（可使用 `gcc -S hook_printf.c` 查看编译得到的汇编指令）：

```
1 gcc -m64 -fPIC --shared hook_printf.c -o hook_printf
```

2.2.3 程序修改

- 安装 `python` 的 `lief` 和 `pwn` 库：

```
1 pip3 install lief
2 pip3 install pwn
```

- 修改脚本文件 `getshell.py` 如下：

```
1 #!/usr/bin/python3
2 from pwn import *
3 import lief
4
5 # Patch a call instruction at srcaddr whose target is
6 # dstaddr.
7 def patch_call(file, srcaddr, dstaddr, arch='amd64'):
```

```

7     print('src address: ', hex(srcaddr), '\ndst address:
    ', hex(dstaddr))
8     relative_offset = p32((dstaddr - (srcaddr + 5)) &
0xffffffffff)
9     call_inst = '\xe8'.encode('latin1') + relative_offset
10    print('call instruction: ', disasm(call_inst,
arch=arch))
11    file.patch_address(srcaddr, [i for i in call_inst])
12
13    binary = lief.parse('./getshell')
14    hook = lief.parse('./hook_printf')
15
16    sec_ehframe = binary.get_section('.eh_frame')
17    print(sec_ehframe)
18    sec_text = hook.get_section('.text')
19    print(sec_text)
20    sec_ehframe.content = sec_text.content
21    print(binary.get_section('.eh_frame'))
22
23    srcaddr = 0x401149
24    dstaddr = sec_ehframe.virtual_address + 0xb9
25
26    patch_call(binary, srcaddr, dstaddr)
27
28    binary.write('getshell.patched')
29    print('patch done!')

```

- 以上脚本中 `srcaddr` 为 `getshell` 中 `call _printf` 指令的地址，`dstaddr` 为 `hook_printf` 中 `newprintf()` 函数的起始地址，即 `.text` 段的起始地址加偏移量 `0xb9`，使用 `patch_call()` 函数构造 `srcaddr` 处向 `newprintf()` 的一条 `call` 指令：

```

.text:000000000401136 ; ===== S U B R O U T I N E =====
.text:000000000401136
.text:000000000401136 ; Attributes: bp-based frame
.text:000000000401136 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:000000000401136 public main
.text:000000000401136 main proc near ; DATA XREF: _start+1Dfo
.text:000000000401136 ; __unwind {
.text:000000000401136 push rbp
.text:000000000401137 mov rbp, rsp
.text:00000000040113A lea rax, format ; "/bin/sh"
.text:000000000401141 mov rdi, rax ; format
.text:000000000401144 mov eax, 0
.text:000000000401149 call _printf
.text:00000000040114E lea rax, s ; "\ngetshell"
.text:000000000401155 mov rdi, rax ; s
.text:000000000401158 call _puts
.text:00000000040115D mov eax, 0
.text:000000000401162 pop rbp
.text:000000000401163 retn
.text:000000000401163 ; } // starts at 401136
.text:000000000401163 main endp
.text:000000000401163
.text:000000000401163 ; -----

```

Name	Start
.LOAD	0000000000000000
.init	0000000000000000
.LOAD	0000000000000017
.plt	0000000000000100
.plt.got	0000000000000100
.LOAD	0000000000000103
.text	0000000000000103
.LOAD	0000000000000117
.fini	0000000000000118
.eh_frame_hdr	0000000000000200
.LOAD	0000000000000204
.eh_frame	0000000000000208
.init_array	0000000000000380
.fini_array	0000000000000388
.LOAD	0000000000000390
.got	00000000000003E0
.got.plt	0000000000000400
.data	0000000000000408
.bss	0000000000000420
.extern	0000000000000428

```

.text:00000000000010F9
.text:00000000000010F9 ; ===== S U B R O U T I N E =====
.text:00000000000010F9
.text:00000000000010F9 ; Attributes: bp-based frame
.text:00000000000010F9
.text:00000000000010F9 public newprintf
.text:00000000000010F9 newprintf ; DATA XREF: LOAD:000000000000300fo
.text:00000000000010F9 ; __unwind {
.text:00000000000010F9 push rbp
.text:00000000000010FA mov rbp, rsp
.text:00000000000010FD mov rsi, 0 ; argv
.text:0000000000001104 mov rdx, 0 ; envp
.text:000000000000110B mov rax, 3Bh
.text:0000000000001112 syscall ; LINUX - sys_execve
.text:0000000000001114 nop
.text:0000000000001115 pop rbp
.text:0000000000001116 retn
.text:0000000000001116 ; // starts at 10F9
.text:0000000000001116 newprintf
.text:0000000000001116 endp
.text:0000000000001116 _text ends
.text:0000000000001116
LOAD:0000000000001117 ; =====

```

- 运行以上脚本并生成 `getshell.patched` 后，尝试运行 `getshell.patched`，但由于 `.eh_frame` 段的权限为只读导致段错误：

```

[12:20:16] xubiang:WORK2 $ ./getshell.py
.eh_frame PROGBITS 402058 100 2058 4.29905 ALLOC LOAD
.text PROGBITS 1040 d7 1040 4.81998 ALLOC EXECINSTR LOAD
.eh_frame PROGBITS 402058 d7 2058 4.81998 ALLOC LOAD
src address: 0x401149
dst address: 0x402111
call instruction: 0: e8 c3 0f 00 00 call 0xfc8
patch done!
[12:20:20] xubiang:WORK2 $ ./getshell.patched
[1] 341952 segmentation fault ./getshell.patched

```

- 为了解决段错误，首先使用 `readelf --segments getshell.patched` 指令查看 `getshell.patched` 的段信息，可知 Section `.eh_frame` 所在的段为 Segment 04：

```

Section to Segment mapping:
Segment Sections...
00
01 .interp
02 .interp .gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn .rela.plt
03 .init .plt .text .fini
04 .rodata .eh_frame_hdr .eh_frame
05 .init_array .fini_array .dynamic .got .got.plt .data .bss
06 .note.gnu.property .note.gnu.build-id .note.ABI-tag
07 .dynamic
08 .note.gnu.property .note.gnu.build-id .note.ABI-tag
09 .note.gnu.property
10 .eh_frame_hdr
11
12 .init_array .fini_array .dynamic .got

```

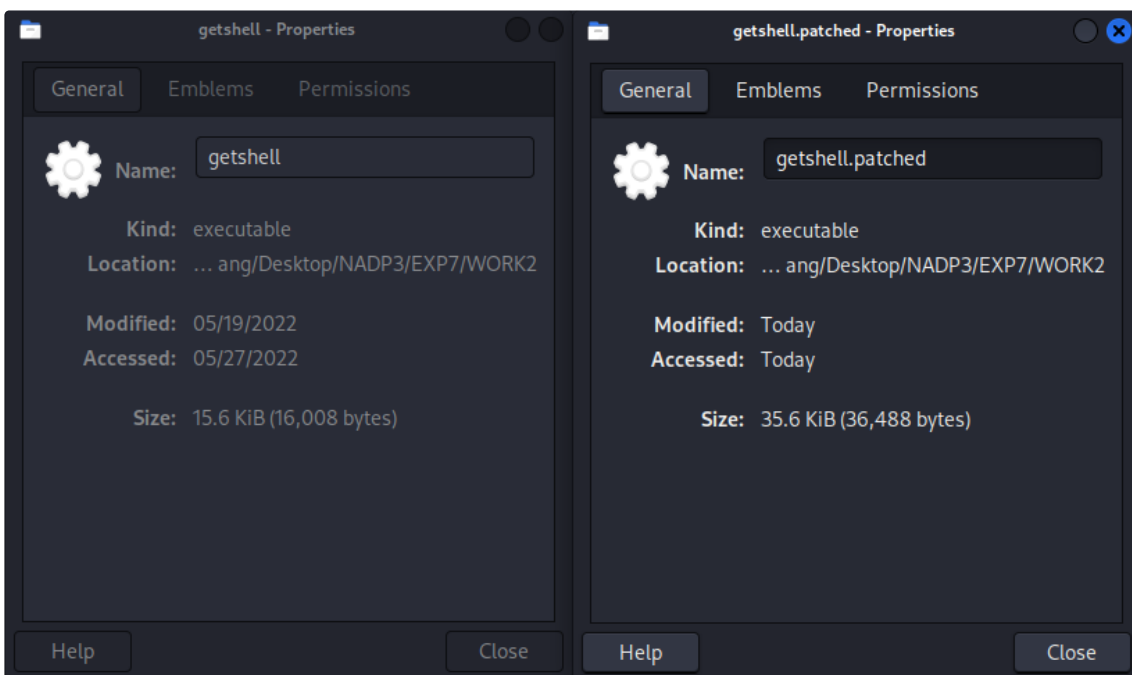
- 安装 `010 Editor`，打开 `getshell.patched` 并导入模板 `ELF.bt` 对该 ELF 文件进行解析，将 Segment 04 对应的权限设为读写执行并保存：

Template Results - ELF.bt						
Name	Value	Start	Size	Color	Comment	
struct file						
struct elf_header		0h	438h	Fg: Bg:		
struct program_header_table		0h	40h	Fg: Bg:	The main elf header basically tel...	
struct program_table_entry64_t program_table_element[0]	(R_) Program Header	11DDh	2D8h	Fg: Bg:	Program headers - describes th...	
struct program_table_entry64_t program_table_element[1]	(R_) Interpreter Path	1215h	38h	Fg: Bg:		
struct program_table_entry64_t program_table_element[2]	(R_) Loadable Segment	124Dh	38h	Fg: Bg:		
struct program_table_entry64_t program_table_element[3]	(R_X) Loadable Segment	1285h	38h	Fg: Bg:		
struct program_table_entry64_t program_table_element[4]	(RWX) Loadable Segment	12BDh	38h	Fg: Bg:		
enum p_type64_e p_type	PT_LOAD (1)	12BDh	4h	Fg: Bg:	Segment type	
enum p_flags64_e p_flags	PF_Read_Write_Exec (7)	12C1h	4h	Fg: Bg:	Segment attributes	
Elf64_Off p_offset_FROM_FILE_BEGIN	2000h	12C5h	8h	Fg: Bg:	Segment offset in file	
Elf64_Addr p_vaddr_VIRTUAL_ADDRESS	0x0000000000402000	12CDh	8h	Fg: Bg:	Segment virtual address	
Elf64_Addr p_paddr_PHYSICAL_ADDRESS	0x0000000000402000	12D5h	8h	Fg: Bg:	Reserved (Segment physical ad...	
Elf64_Xword p_filesz_SEGMENT_FILE_LENGTH	344	12DDh	8h	Fg: Bg:	Segment size in file	
Elf64_Xword p_memsz_SEGMENT_RAM_LENGTH	344	12E5h	8h	Fg: Bg:	Segment size in ram	
Elf64_Xword p_align	4096	12EDh	8h	Fg: Bg:	Segment alignment	

- 再次尝试运行 `getshell.patched`，可以正常获得 `shell` 并完成相关操作（获得的 `shell` 的权限与执行程序的用户权限、程序的 `setuid` 位、`/bin/sh` 链接的 `shell` 程序等因素有关，不再深入探究，此处使用 `root` 权限运行程序以获得具有 `root` 权限的 `shell`）：

```
File Actions Edit View Help
xubiang@kali:~/Desktop/NADP3/EXP7/WORK2
[12:37:41] xubiang:WORK2 $ sudo ./getshell.patched
# whoami
root
# id
uid=0(root) gid=0(root) groups=0(root),4(adm),20(dialout),119(wireshark),142(kaboxer)
# echo "U201911803" > num.txt
# cat num.txt
U201911803
# exit
[12:38:14] xubiang:WORK2 $
```

- 使用这种方式得到的程序大小有变化，原程序大小为 **16008 Bytes**，修改后的程序大小为 **36,488 Bytes**，这是 **LIEF** 库导致的：



2.2.4 手动插入补丁代码

- 将以上生成的 `newprintf()` 函数中的汇编代码手动插入到程序的 `.eh_frame` 段即可在不改变程序大小的前提下完成补丁，使用以上使用 **IDA** 查看的 `getshell` 的结果可知 `call _printf` 指令的地址为 `0x401149`，吓一跳指令的地址为 `0x40114E`，需要插入的汇编代码如下（省略了函数调用的相关指令）：

```
1 | ; rdi 在原程序中已为 /bin/sh 的地址
2 | mov rsi, 0      ; rsi ← NULL
3 | mov rdx, 0      ; rdi ← NULL
4 | mov rax, 59     ; rax ← 59 (sys_execve)
5 | syscall
6 | jmp 0x40114E    ; jmp back
```

- 在 **IDA** 中查看原程序的段表，并查看 `.eh_frame` 段，选择 `0x402088` 处作为补丁代码的起始位置：

Name	Start
LOAD	0000000000400000
.init	0000000000401000
LOAD	0000000000401017
.plt	0000000000401020
.text	0000000000401050
LOAD	00000000004011D1
.fini	00000000004011D4
.rodata	0000000000402000
LOAD	0000000000402016
.eh_frame_hdr	0000000000402018
LOAD	0000000000402054
.eh_frame	0000000000402058
.init_array	0000000000403E10
.fini_array	0000000000403E18
LOAD	0000000000403E20
.got	0000000000403FF0
.got.plt	0000000000404000
.data	0000000000404028
.bss	0000000000404038
.prgrend	0000000000404040
extern	0000000000404048

.eh_frame:0000000000402086	db	0
.eh_frame:0000000000402087	db	0
.eh_frame:0000000000402088	db	0
.eh_frame:0000000000402089	db	0
.eh_frame:000000000040208A	db	0
.eh_frame:000000000040208B	db	0
.eh_frame:000000000040208C	db	1
.eh_frame:000000000040208D	db	7Ah ; z
.eh_frame:000000000040208E	db	52h ; R
.eh_frame:000000000040208F	db	0
.eh_frame:0000000000402090	db	1
.eh_frame:0000000000402091	db	78h ; x
.eh_frame:0000000000402092	db	10h
.eh_frame:0000000000402093	db	1
.eh_frame:0000000000402094	db	1Bh
.eh_frame:0000000000402095	db	0Ch
.eh_frame:0000000000402096	db	7

- 将补丁的汇编代码使用 **Keypatch** 的 **Patcher** 或 **Fill range** 功能填充到原程序 **.eh_frame** 段的相应位置，点击 **C** 将其转换为代码：

```
.eh_frame:0000000000402088 ; -----
.eh_frame:0000000000402088      mov     rsi, 0           ; Keypatch modified this from:
.eh_frame:0000000000402088      ; db 0
.eh_frame:0000000000402088      ; db 0
.eh_frame:0000000000402088      ; db 0
.eh_frame:0000000000402088      ; db 0
.eh_frame:0000000000402088      ; db 1
.eh_frame:0000000000402088      ; db 7Ah
.eh_frame:0000000000402088      ; db 52h
.eh_frame:000000000040208F      mov     rdx, 0           ; Keypatch modified this from:
.eh_frame:000000000040208F      ; db 0
.eh_frame:000000000040208F      ; db 1
.eh_frame:000000000040208F      ; db 78h
.eh_frame:000000000040208F      ; db 10h
.eh_frame:000000000040208F      ; db 1
.eh_frame:000000000040208F      ; db 1Bh
.eh_frame:000000000040208F      ; db 0Ch
.eh_frame:0000000000402096      mov     rax, 3Bh        ; Keypatch modified this from:
.eh_frame:0000000000402096      ; db 7
.eh_frame:0000000000402096      ; db 8
.eh_frame:0000000000402096      ; db 90h
.eh_frame:0000000000402096      ; db 1
.eh_frame:0000000000402096      ; db 0
.eh_frame:0000000000402096      ; db 0
.eh_frame:0000000000402096      ; db 10h
.eh_frame:000000000040209D      syscall          ; Keypatch modified this from:
.eh_frame:000000000040209D      ; db 0
.eh_frame:000000000040209D      ; db 0
.eh_frame:000000000040209F      jmp     loc_40114E    ; Keypatch modified this from:
.eh_frame:000000000040209F      ; db 0
.eh_frame:000000000040209F      ; db 1Ch
.eh_frame:000000000040209F      ; db 0
.eh_frame:000000000040209F      ; db 0
.eh_frame:000000000040209F      ; db 0
.eh_frame:000000000040209F ; -----
```

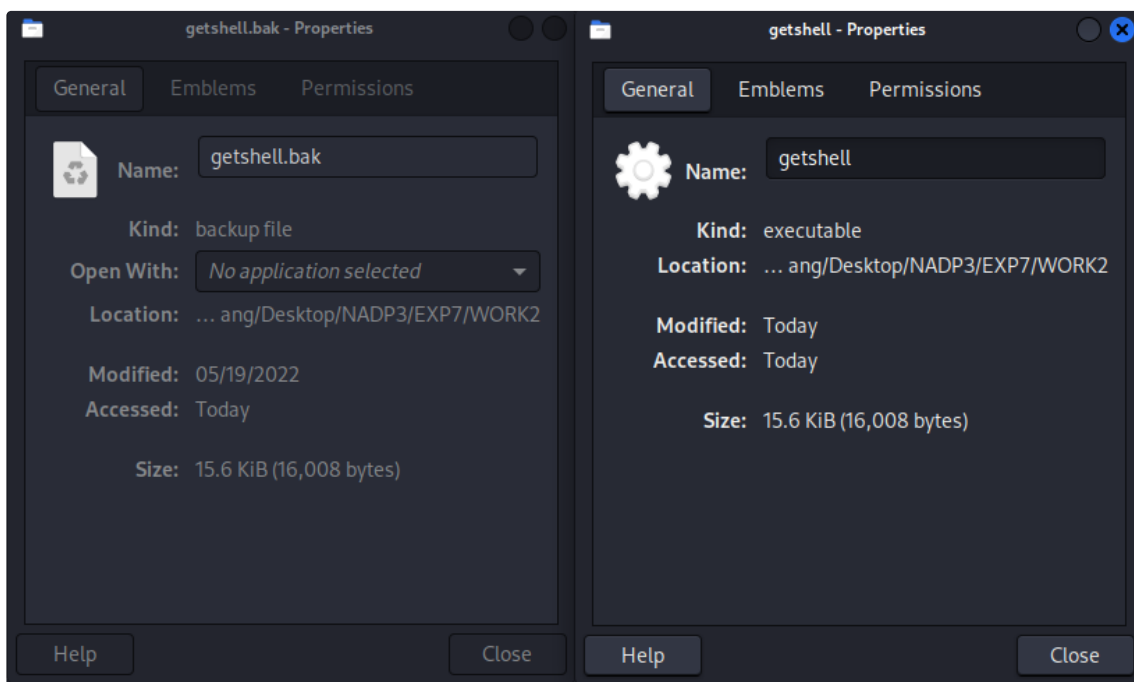
- 再将源程序的 **call _printf** 指令使用 **Keypatch** 修改为 **jmp 0x402088** 即可：

```
.text:0000000000401136 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:0000000000401136      public main
.text:0000000000401136      main          proc near          ; DATA XREF: _start+1D10
.text:0000000000401136      ; FUNCTION CHUNK AT .eh_frame:0000000000402088 SIZE 0000001C BYTES
.text:0000000000401136      ; __unwind {
.text:0000000000401136      push     rbp
.text:0000000000401137      mov     rbp, rsp
.text:000000000040113A      lea     rax, filename ; "/bin/sh"
.text:0000000000401141      mov     rdi, rax      ; filename
.text:0000000000401144      mov     eax, 0
.text:0000000000401149      jmp     loc_402088    ; Keypatch modified this from:
.text:0000000000401149      ; call _printf
.text:000000000040114E ; -----
.text:000000000040114E      loc_40114E:          ; CODE XREF: main+F694j
.text:000000000040114E      lea     rax, s          ; "\ngetshell"
.text:0000000000401155      mov     rdi, rax      ; s
.text:0000000000401158      call    _puts
.text:000000000040115D      mov     eax, 0
.text:0000000000401162      pop     rbp
.text:0000000000401163      retn
.text:0000000000401163 ; } // starts at 401136
.text:0000000000401163      main          endp
```

- 将以上修改应用到源文件并使用上述修改段属性的方法修改 `.eh_frame` 段的权限后进行测试，可以正常完成功能：

```
xubiang@kali:~/Desktop/NADP3/EXP7/WORK2
[13:16:08] xubiang:WORK2 $ sudo ./getshell
# whoami
root
# id
uid=0(root) gid=0(root) groups=0(root),4(adm),20(dialout),119(wireshark),142(kaboxer)
# echo "U201911803" > num2.txt
# cat num2.txt
U201911803
# exit
[13:16:27] xubiang:WORK2 $
```

- 查看该修改后的程序 `getshell` 与原程序 `getshell.bak` 的大小均为 16008 Bytes，大小没有发生变化：



2.2.5 使用 Preload Hook

- 使用 Preload Hook 可以不对程序本身做任何修改，因此也不会改变程序大小，编写自定义的 `printf()` 函数如下：

```
1 // preload_hook_printf.c
2 #define _GNU_SOURCE
3 #include <sys/stat.h>
4 #include <unistd.h>
5 #include <dlfcn.h>
6 #include <string.h>
7 #include <stdlib.h>
8
9 int printf(char *a, int b) {
10     if (!strcmp(a, "/bin/sh")) {
11         system(a);
```



```

12     }
13     int (*old_printf)(char *, int);
14     old_printf = (int (*)(char *, int))dlsym(RTLD_NEXT,
15     "printf");
16     old_printf(a, b);
17     puts("\n");
18 }

```

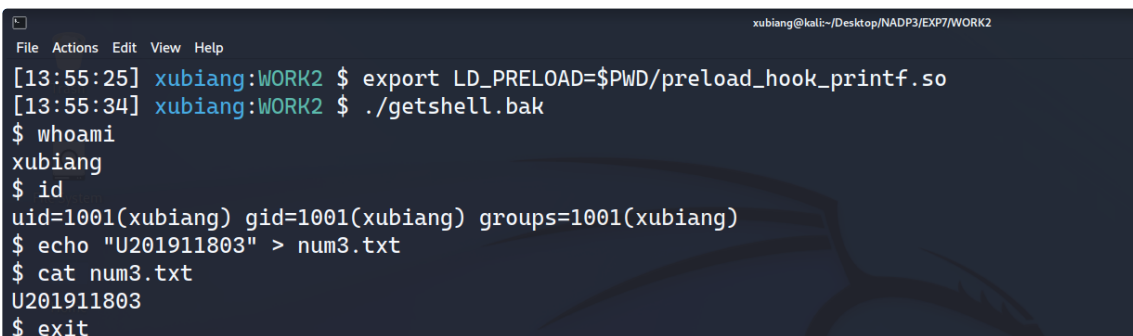
- 编译该文件生成动态链接库：

```

1 gcc -m64 -fPIC --shared preload_hook_printf.c -o
   preload_hook_printf.so -ldl

```

- 加载动态链接库 `export LD_PRELOAD=$PWD/preload_hook_printf.so`，并执行 `getshell.bak` 程序，可以正常完成功能：



```

xubiang@kali:~/Desktop/NADP3/EXP7/WORK2
File Actions Edit View Help
[13:55:25] xubiang:WORK2 $ export LD_PRELOAD=$PWD/preload_hook_printf.so
[13:55:34] xubiang:WORK2 $ ./getshell.bak
$ whoami
xubiang
$ id
uid=1001(xubiang) gid=1001(xubiang) groups=1001(xubiang)
$ echo "U201911803" > num3.txt
$ cat num3.txt
U201911803
$ exit

```

- 由于未对 `getshell.bak` 程序做任何更改，其大小未发生改变，仍为 **16008 Bytes**：

