综合实践 3

二进制文件补丁技术 实验手册

一、实验目的

掌握二进制文件补丁技术的原理及常用方法，能够根据需要对存在漏洞的二进制程序进行修补。

二、实验内容

1、简单修改二进制文件实现漏洞修补；

2、插入补丁代码实现漏洞修补

3、利用 LIEF 库实现漏洞修补

4、热补丁技术

三、简单修改二进制文件实现漏洞修补

1、格式化字符串漏洞 print_with_puts

漏洞源码，注意划线处：

```c
#include<stdio.h>

int main() {
        puts("test1");
        char s[20];
        scanf("%s", s);
        printf(s);
        return 0;
}
```

编译：

```
root@DESKTOP-HUI9I31:/# gcc print_with_puts.c -o print_with_puts
```

漏洞验证：

```
root@DESKTOP-HUI9I31:/# ./print_with_puts
test1
%s%s%s%s%s%s%s%s%s%s%s%s
段错误 (核心已转储)
```

在只有二进制文件而没有源代码的情况下，用 IDA 进行反编译，注意划线处：

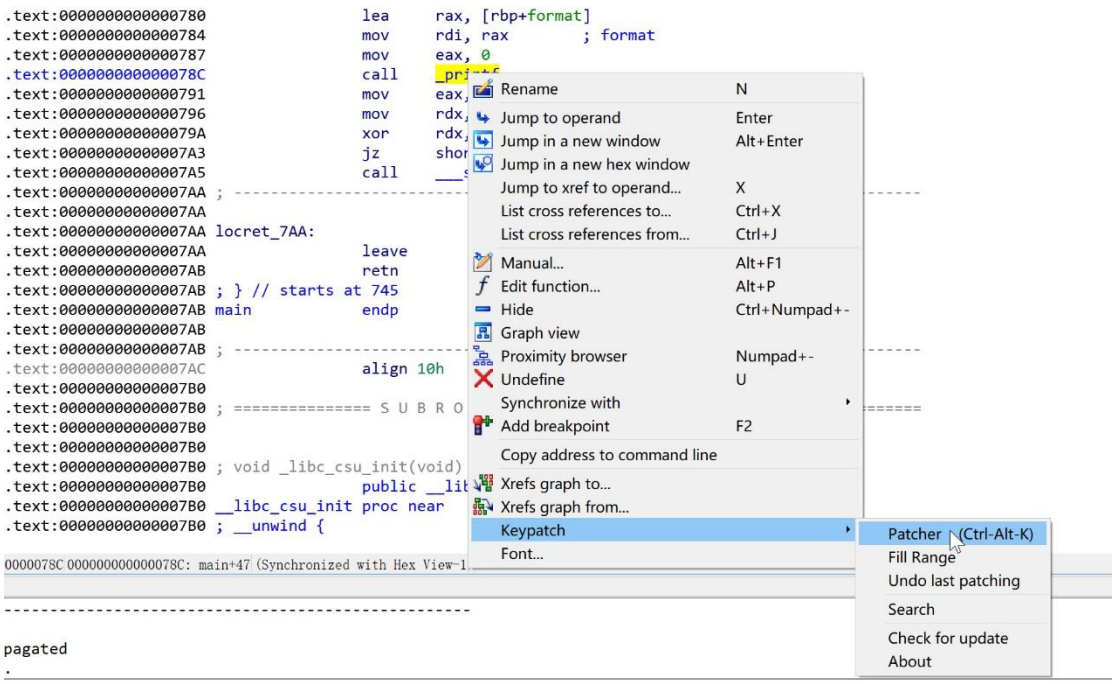
```
.text:0000000000000745                  push    rbp
.text:0000000000000746                  mov     rbp, rsp
.text:0000000000000749                  sub     rsp, 20h
.text:000000000000074D                  mov     rax, fs:28h
.text:0000000000000756                  mov     [rbp+var_8], rax
.text:000000000000075A                  xor     eax, eax
.text:000000000000075C                  lea     rdi, s          ; "test1"
.text:0000000000000763                  call    _puts
.text:0000000000000768                  lea     rax, [rbp+format]
.text:000000000000076C                  mov     rsi, rax
.text:000000000000076F                  lea     rdi, aS         ; "%s"
.text:0000000000000776                  mov     eax, 0
.text:000000000000077B                  call    ___isoc99_scanf
.text:0000000000000780                  lea     rax, [rbp+format]
.text:0000000000000784                  mov     rdi, rax        ; format
.text:0000000000000787                  mov     eax, 0
.text:000000000000078C                  call    _printf
.text:0000000000000791                  mov     eax, 0
.text:0000000000000796                  mov     rdx, [rbp+var_8]
```

观察到该程序中存在 puts 函数，且调用 puts 函数与调用 printf 函数的指令均为五个字节，想到可以将 call _printf 简单修改为 call _puts，方法如下：

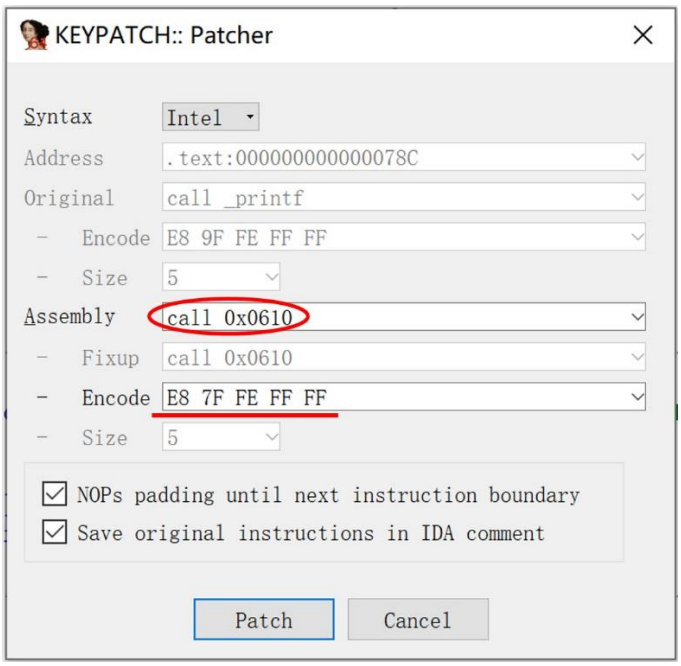
(1)查看 puts 函数的地址：结果为 0x0610

```
.plt:0000000000000610 _puts           proc near               ; CODE XREF: main+1E↓p
.plt:0000000000000610                 jmp     cs:puts_ptr
.plt:0000000000000610 _puts           endp
```

(2)选中调用 printf 指令的语句，通过鼠标右键或 Ctrl+Alt+K 快捷键调用 IDA 插件 Keypatch：



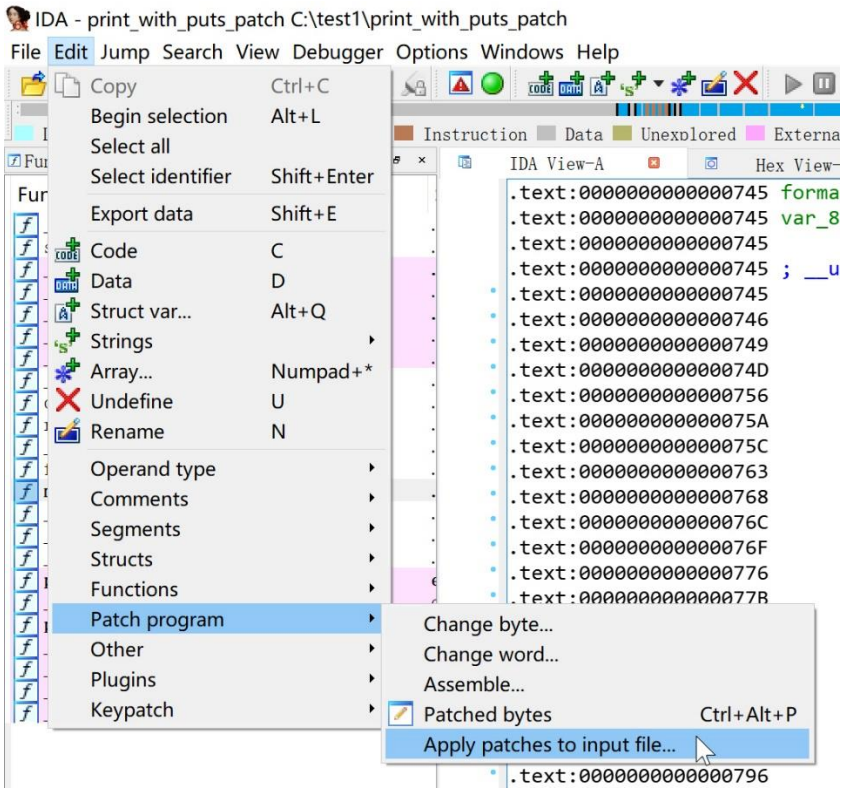
(3)将调用 printf 函数的语句修改为调用 puts 函数的语句,注意,由于 Keypatch 不能识别符号地址跳转，因此修改时不能使用 call _puts 这样的语句，而应该直接给定跳转地址，这也是第(1)步中必须准备好 puts 函数地址的原因：



Keypatch 修改后的结果为：

```
.text:000000000000075A                  xor     eax, eax
.text:000000000000075C                  lea     rdi, s          ; "test1"
.text:0000000000000763                  call    _puts
.text:0000000000000768                  lea     rax, [rbp+format]
.text:000000000000076C                  mov     rsi, rax
.text:000000000000076F                  lea     rdi, aS         ; "%s"
.text:0000000000000776                  mov     eax, 0
.text:000000000000077B                  call    ___isoc99_scanf
.text:0000000000000780                  lea     rax, [rbp+format]
.text:0000000000000784                  mov     rdi, rax        ; s
.text:0000000000000787                  mov     eax, 0
.text:000000000000078C                  call    _puts           ; Keypatch modified this from:
.text:000000000000078C                                          ;   call _printf
.text:0000000000000791                  mov     eax, 0
.text:0000000000000796                  mov     rdx, [rbp+var_8]
```

(4)通过 IDA 将 Keypatch 的修改结果保存到二进制文件：



补丁验证：

```
root@DESKTOP-HUI9I31:/# ./print_with_puts_patch
test1
%s%s%s%s%s%s%s%s%s%s%s%s
%s%s%s%s%s%s%s%s%s%s%s%s
```

四、在.eh_frame 段插入补丁代码实现漏洞修补

1、格式化字符串漏洞 print_with_printf

漏洞源码，注意划线处：

```
#include <stdio.h>

int main() {
        printf("test2.1");
        char s[10];
        scanf("%s", s);
        printf(s);
        return 0;
}
```

编译：

```
root@DESKTOP-HUI9I31:/# gcc print_with_printf.c -o print_with_printf
```

漏洞验证：

```
%proot@DESKTOP-HUI9I31:/# ./print_with_printf
test2.1%p
0xaroot@DESKTOP-HUI9I31:/#
root@DESKTOP-HUI9I31:/#
```

与实验一类似，用 IDA 进行反编译，注意划线处：

```
.text:000000000000071A                push    rbp
.text:000000000000071B                mov     rbp, rsp
.text:000000000000071E                sub     rsp, 20h
.text:0000000000000722                mov     rax, fs:28h
.text:000000000000072B                mov     [rbp+var_8], rax
.text:000000000000072F                xor     eax, eax
.text:0000000000000731                lea     rdi, format     ; "test2.1"
.text:0000000000000738                mov     eax, 0
.text:000000000000073D                call    _printf
.text:0000000000000742                lea     rax, [rbp+format]
.text:0000000000000746                mov     rsi, rax
.text:0000000000000749                lea     rdi, aS         ; "%s"
.text:0000000000000750                mov     eax, 0
.text:0000000000000755                call    ___isoc99_scanf
.text:000000000000075A                lea     rax, [rbp+format]
.text:000000000000075E                mov     rdi, rax        ; format
.text:0000000000000761                mov     eax, 0
.text:0000000000000766                call    _printf
.text:000000000000076B                mov     eax, 0
.text:0000000000000770                mov     rdx, [rbp+var_8]
```

与实验一不同，该程序中不存在 puts 函数，因此实验一中的漏洞修补方法不能用于该程序。观察到程序调用 scanf 函数时参数引用正确，不存在格式化字符串漏洞，想到可以修改存在漏洞的 printf 函数调用，修改后的函数调用可以写入程序的.eh_frame 段，方法如下：

(1)模仿 scanf 函数的调用写出补丁代码：

```
mov     rsi, rdi
lea     rdi, "%s"
call    _printf
```

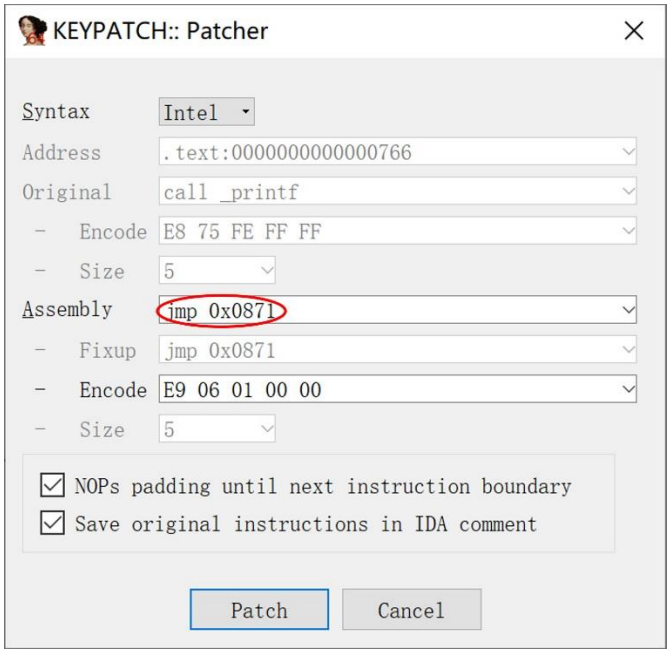
(2)查看 printf 函数的地址：结果为 0x05E0

```
.plt:00000000000005E0 _printf         proc near               ; CODE XREF: main+23↓p
.plt:00000000000005E0                                         ; main+4C↓p
.plt:00000000000005E0                 jmp     cs:printf_ptr
.plt:00000000000005E0 _printf         endp
```

(3)查看程序的.eh_frame 段，寻找可以写入补丁代码的空间：假设选择 0x0871 到 0x088F 作为存放补丁代码的空间

```
.eh_frame:0000000000000870                 db  1Bh
.eh_frame:0000000000000871                 db  0Ch
.eh_frame:0000000000000872                 db    7
.eh_frame:0000000000000873                 db    8
.eh_frame:0000000000000874                 db  90h
.eh_frame:0000000000000875                 db    1
.eh_frame:0000000000000876                 db    7
.eh_frame:0000000000000877                 db  10h
.eh_frame:0000000000000878                 db  14h
.eh_frame:0000000000000879                 db    0
.eh_frame:000000000000087A                 db    0
.eh_frame:000000000000087B                 db    0
.eh_frame:000000000000087C                 db  1Ch
.eh_frame:000000000000087D                 db    0
.eh_frame:000000000000087E                 db    0
.eh_frame:000000000000087F                 db    0
.eh_frame:0000000000000880                 db  90h
.eh_frame:0000000000000881                 db 0FDh
.eh_frame:0000000000000882                 db 0FFh
.eh_frame:0000000000000883                 db 0FFh
.eh_frame:0000000000000884                 db  2Bh ; +
.eh_frame:0000000000000885                 db    0
.eh_frame:0000000000000886                 db    0
.eh_frame:0000000000000887                 db    0
.eh_frame:0000000000000888                 db    0
.eh_frame:0000000000000889                 db    0
.eh_frame:000000000000088A                 db    0
.eh_frame:000000000000088B                 db    0
.eh_frame:000000000000088C                 db    0
.eh_frame:000000000000088D                 db    0
.eh_frame:000000000000088E                 db    0
.eh_frame:000000000000088F                 db    0
.eh_frame:0000000000000890                 db  14h
```

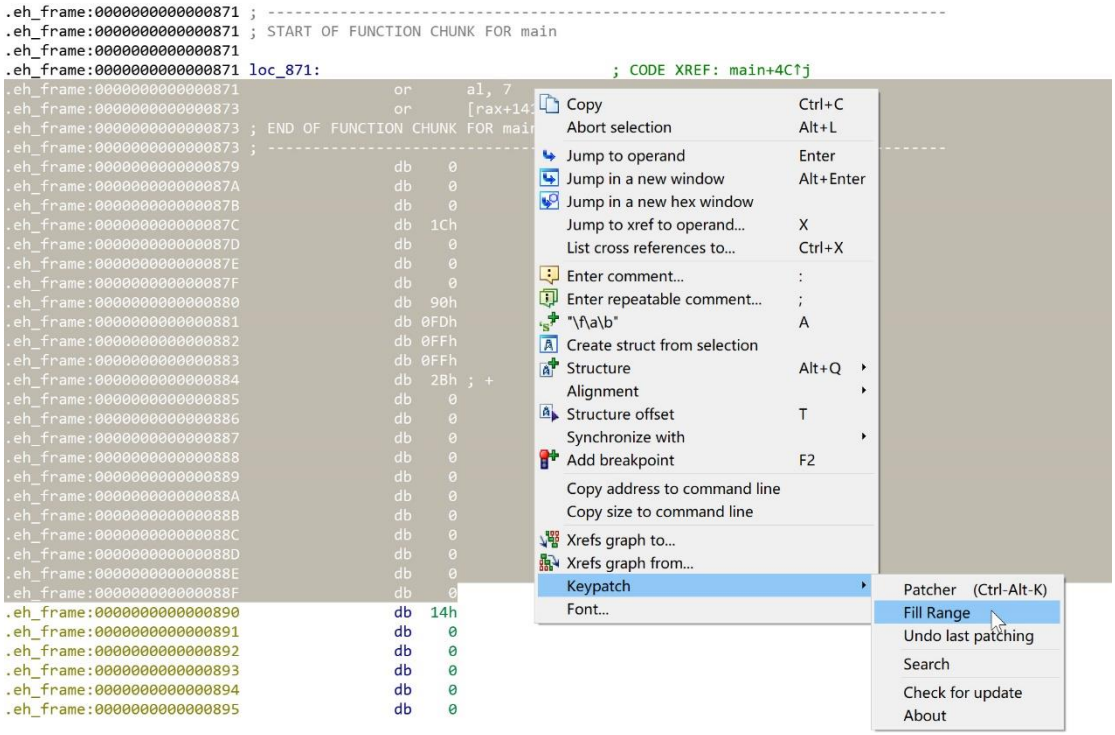
(4)将存在漏洞的 printf 函数调用语句修改为跳转语句，跳转到.eh_frame 段的补丁代码处执行，并查看其下一条语句的地址：



Keypatch 修改后的结果为：

```
.text:000000000000072F                  xor     eax, eax
.text:0000000000000731                  lea     rdi, format     ; "test2.1"
.text:0000000000000738                  mov     eax, 0
.text:000000000000073D                  call    _printf
.text:0000000000000742                  lea     rax, [rbp+format]
.text:0000000000000746                  mov     rsi, rax
.text:0000000000000749                  lea     rdi, aS         ; "%s"
.text:0000000000000750                  mov     eax, 0
.text:0000000000000755                  call    ___isoc99_scanf
.text:000000000000075A                  lea     rax, [rbp+format]
.text:000000000000075E                  mov     rdi, rax        ; format
.text:0000000000000761                  mov     eax, 0
.text:0000000000000766                  jmp     loc_871         ; Keypatch modified this from:
.text:0000000000000766                                          ;   call _printf
.text:000000000000076B ; ---------------------------------------------------------------------------
.text:000000000000076B                  mov     eax, 0
.text:0000000000000770                  mov     rdx, [rbp+var_8]
```

(5)根据前面查看的 printf 函数地址和下一条语句地址修改补丁代码，并将补丁代码写入.eh_frame 段：



Keypatch 修改后的结果为：

```
.eh_frame:0000000000000871
.eh_frame:0000000000000871 loc_871:                        ; CODE XREF: main+4C↑j
.eh_frame:0000000000000871                 mov     rsi, rdi        ; Keypatch filled range [0x871:0x88E] (30 bytes), replaced:
.eh_frame:0000000000000871                                         ;   or al, 7
.eh_frame:0000000000000871                                         ;   db 8
.eh_frame:0000000000000871                                         ;   nop
.eh_frame:0000000000000871                                         ;   add [rdi], eax
.eh_frame:0000000000000871                                         ;   adc [rax+rax], dl
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000871                                         ;   add [rax+rax], bl
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000871                                         ;   nop
.eh_frame:0000000000000871                                         ;   std
.eh_frame:0000000000000871                                         ;   db 0FFh
.eh_frame:0000000000000871                                         ;   db 0FFh
.eh_frame:0000000000000871                                         ;   db 2Bh
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000871                                         ;   db 0
.eh_frame:0000000000000874                 lea     rdi, aS         ; "%s"
.eh_frame:000000000000087B                 call    _printf
.eh_frame:000000000000087B ; END OF FUNCTION CHUNK FOR main
.eh_frame:0000000000000880                 jmp     loc_76B
.eh_frame:0000000000000880 ; ---------------------------------------------------------------------------
.eh_frame:0000000000000885                 db  90h
.eh_frame:0000000000000886                 db  90h
.eh_frame:0000000000000887                 db  90h
.eh_frame:0000000000000888                 db  90h
.eh_frame:0000000000000889                 db  90h
.eh_frame:000000000000088A                 db  90h
```

(6)与实验一类似地，通过 IDA 将 Keypatch 的修改结果保存到二进制文件即可。

补丁验证：

```
root@DESKTOP-HUI9I31:/# ./print_with_printf_patch
test2.1%p
%proot@DESKTOP-HUI9I31:/#
root@DESKTOP-HUI9I31:/#
```

2、释放重引用漏洞 use_after_free

漏洞源码，注意划线处：

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

typedef struct obj {
        char *name;
        void (*func)(char *str);
} OBJ;

void obj_func(char *str) {
        printf("%s\n", str);
}

void fake_obj_func() {
        printf("This is a fake obj_func which means uaf vul.\n");
}

int main() {
        OBJ *obj_1;
        obj_1 = (OBJ *)malloc(sizeof(struct obj));
        obj_1->name = "obj_name";
        obj_1->func = obj_func;
        obj_1->func("This is an object.");
        free(obj_1);
        printf("The objcet has been freed, use it again will lead crash.\n");
        obj_1->func("But it seems that...");
        obj_1->func = fake_obj_func;
        obj_1->func("Try again.\n");
}
```

编译：

```
root@DESKTOP-HUI9I31:/# gcc use_after_free.c -o use_after_free
```

漏洞验证：

```
root@DESKTOP-HUI9I31:/# ./use_after_free
This is an object.
The objcet has been freed, use it again will lead crash.
But it seems that...
This is a fake obj_func which means uaf vul.
root@DESKTOP-HUI9I31:/#
```

用 IDA 进行反编译，注意划线处：

```
.text:00000000000006F8                push    rbp
.text:00000000000006F9                mov     rbp, rsp
.text:00000000000006FC                sub     rsp, 10h
.text:0000000000000700                mov     edi, 10h        ; size
.text:0000000000000705                call    _malloc
.text:000000000000070A                mov     [rbp+ptr], rax
.text:000000000000070E                mov     rax, [rbp+ptr]
.text:0000000000000712                lea     rdx, aObjName   ; "obj_name"
.text:0000000000000719                mov     [rax], rdx
.text:000000000000071C                mov     rax, [rbp+ptr]
.text:0000000000000720                lea     rdx, obj_func
.text:0000000000000727                mov     [rax+8], rdx
.text:000000000000072B                mov     rax, [rbp+ptr]
.text:000000000000072F                mov     rax, [rax+8]
.text:0000000000000733                lea     rdi, aThisIsAnObject ; "This is an object."
.text:000000000000073A                call    rax
.text:000000000000073C                mov     rax, [rbp+ptr]
.text:0000000000000740                mov     rdi, rax        ; ptr
.text:0000000000000743                call    _free
.text:0000000000000748                lea     rdi, aTheObjcetHasBe ; "The objcet has been freed, use it again"...
.text:000000000000074F                call    _puts
.text:0000000000000754                mov     rax, [rbp+ptr]
.text:0000000000000758                mov     rax, [rax+8]
.text:000000000000075C                lea     rdi, aButItSeemsThat ; "But it seems that..."
.text:0000000000000763                call    rax
.text:0000000000000765                mov     rax, [rbp+ptr]
.text:0000000000000769                lea     rdx, fake_obj_func
.text:0000000000000770                mov     [rax+8], rdx
.text:0000000000000774                mov     rax, [rbp+ptr]
.text:0000000000000778                mov     rax, [rax+8]
.text:000000000000077C                lea     rdi, aTryAgain  ; "Try again.\n"
.text:0000000000000783                call    rax
.text:0000000000000785                mov     eax, 0
```

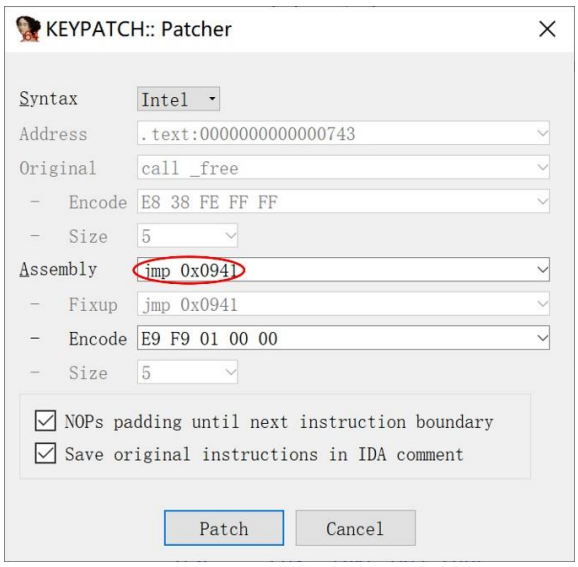
考虑 UAF 漏洞的成因，主要是调用 free 函数释放对象时，没有将指向该对象的指针置为 0，导致产生可被恶意调用的悬垂指针。修补时只需要在调用 free 函数的同时将对象指针置为 0 即可，补丁代码可以写入程序的.eh_frame 段，方法如下：

(1)查看 free 函数的地址：结果为 0x0580

```
.plt:0000000000000580 _free           proc near               ; CODE XREF: main+4B↓p
.plt:0000000000000580                 jmp     cs:free_ptr
.plt:0000000000000580 _free           endp
```

(2)查看程序的.eh_frame 段，寻找可以写入补丁代码的空间：假设选择 0x0941 到 0x0960 作为存放补丁代码的空间

(3)将 free 函数调用语句修改为跳转语句，跳转到.eh_frame 段中补丁所在处，并查看其下一条语句的地址：



Keypatch 修改后的结果为：

```
.text:0000000000000733                lea     rdi, aThisIsAnObject ; "This is an object."
.text:000000000000073A                call    rax
.text:000000000000073C                mov     rax, [rbp+ptr]
.text:0000000000000740                mov     rdi, rax        ; ptr
.text:0000000000000743                jmp     near ptr unk_941 ; Keypatch modified this from:
.text:0000000000000743                                         ;   call _free
.text:0000000000000748 ; ---------------------------------------------------------------------------
.text:0000000000000748                lea     rdi, aTheObjcetHasBe ; "The objcet has been freed, use it again"...
```

(4)根据前面查看的 free 函数地址和下一条语句地址写出补丁代码，并将其写入.eh_frame 段，修改后的结果为：

```
.eh_frame:0000000000000941 loc_941:                                ; CODE XREF: main+4B↑j
.eh_frame:0000000000000941                mov     [rbp+ptr], 0
.eh_frame:0000000000000949                call    _free
.eh_frame:000000000000094E                jmp     loc_748
.eh_frame:000000000000094E ; END OF FUNCTION CHUNK FOR main
.eh_frame:000000000000094E ; ---------------------------------------------------------------------------
.eh_frame:0000000000000953                db  90h
.eh_frame:0000000000000954                db  90h
.eh_frame:0000000000000955                db  90h
.eh_frame:0000000000000956                db  90h
.eh_frame:0000000000000957                db  90h
.eh_frame:0000000000000958                db  90h
.eh_frame:0000000000000959                db  90h
.eh_frame:000000000000095A                db  90h
.eh_frame:000000000000095B                db  90h
.eh_frame:000000000000095C                db  90h
.eh_frame:000000000000095D                db  90h
.eh_frame:000000000000095E                db  90h
.eh_frame:000000000000095F                db  90h
```

(5)通过 IDA 将 Keypatch 的修改结果保存到二进制文件即可。补丁验证：

```
root@DESKTOP-HUI9I31:/# ./use_after_free_patch
This is an object.
The objcet has been freed, use it again will lead crash.
段错误 (核心已转储)
root@DESKTOP-HUI9I31:/#
```

五、利用 LIEF 库实现漏洞修补

LIEF 是一个开源的跨平台的可执行文件修改工具，它能够解析 ELF、PE 等二进制程序文件，并提供一个用户友好的 API 来将一个二进制程序中的机器码写到另一个二进制程序中，从而方便地实现补丁编写和漏洞修补。LIEF 对外提供了 Python、C++、C 的编程接口，下面以 Python 接口为例来进行实验。

1、使用 LIEF 增加 segment 实现漏洞修补

实验程序的源代码如下：

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char** argv) {
  printf("/bin/sh%d",102);
  puts("let's go\n");
  printf("/bin/sh%d",102);
  puts("let's gogo\n");
  return EXIT_SUCCESS;
}
```

编写一个包含补丁代码的静态函数库，将 printf 函数修改为一个新的“补丁”函数 write(0,”/bin/sh%d”,0x20)：

```
void myprintf(char *a,int b){
        asm(
                "mov %rdi,%rsi\n"
                "mov $0,%rdi\n"
                "mov $0x20,%rdx\n"
                "mov $0x1,%rax\n"
                "syscall\n"
                );
}
void myputs(char *a){
        asm(
                "push $0x41414141\n"
                "push $0x42424242\n"
                "push %rsp\n"
                "pop  %rsi\n"
                "mov $0,%rdi\n"
                "mov $0x20,%rdx\n"
                "mov $0x1,%rax\n"
                "syscall\n"
                "pop %rax\n"
                "pop %rax\n"
                );
}
```

利用 LIEF 提供的 add 参数为二进制文件增加 segment，segment 的内容就是上面的补丁代码：

```
binary    = lief.parse(binary_name)
lib  = lief.parse(lib_name)
segment_add = binary.add(lib.segments[0])
```

修改跳转逻辑，将 call printf 改为 call myprintf，由于 call 指令的寻址方式是相对寻址，即 call addr = EIP + addr，因此需要计算写入的新函数距离要修改指令的偏移，计算方法如下：

```
call xxx=(addr of new segment + offset function ) - (addr of order + 5/*length of call xx*/)
```

由于偏移地址是补码表示的，因此计算时需要对结果异或 0xffffffff，最终的 LIEF 脚本如下：

```
def patch_call(file,where,end,arch = "amd64"):
        print hex(end)
        length = p32((end - (where + 5 )) & 0xffffffff)
        order = '\xe8'+length
        print disasm(order,arch=arch)
        file.patch_address(where,[ord(i) for i in order])
```

执行上面的脚本之后可以看到 patch 成功：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  sub_8022F9("/bin/sh%d");
  puts("let's go\n");
  printf("/bin/sh%d", 102LL, argv);
  puts("let's gogo\n");
  return 0;
}
```

```
__int64 __fastcall sub_8022F9(const char *buf)
{
  __int64 result; // rax

  result = 1LL;
  __asm { syscall; LINUX - sys_write }
  return result;
}
```

2、使用 LIEF 增加 library 实现漏洞修补

实验程序的源代码如下：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  printf("/bin/sh%d", 102LL, envp, argv);
  return 0;
}
```

编写一个包含补丁代码的动态链接库，将原来的 printf 函数修改为一个自定义的 printf 函数：

```
#define _GNU_SOURCE
//#include <stdio.h>
#include <sys/stat.h>
#include <unistd.h>
#include <dlfcn.h>

int printf(char *a,int b) {
        char str[] = "hacked by me\n ";
        //puts(a);
        if(strstr(a,"/bin/sh")){
                puts("find dangerous str~");
        }
        int (*old_printf)(char *,int);
        old_printf =(int (*)(char *,int)) dlsym(RTLD_NEXT, "printf");
        old_printf(a,b);
        puts("\n");
}
```

由于当程序中加载了两个 library，而其所调用的某个函数在两个 library 内同名存在时，此函数是按照一定的顺序在不同的 library 中进行查找的，因此这里可以在不修改程序正常代码的前提下实现对 printf 函数的 hook，进而实现补丁的修补。

3、使用 LIEF 修改.eh_frame 段实现漏洞修补

section 对象中的 content 属性就是该 section 的内容，因此，要修改程序的.eh_frame 段，写入补丁代码，只需将补丁程序中的.text 段赋值到.eh_frame 段即可。赋值完成后，通过与前面相同的方法修改函数跳转地址，使漏洞程序跳转到.eh_frame 段来执行补丁代码。最终的 LIEF 脚本如下：

```python
import lief
from pwn import *

def patch_call(file,srcaddr,dstaddr,arch = "amd64"):
        print hex(dstaddr)
        length = p32((dstaddr - (srcaddr + 5 )) & 0xffffffff)
        order = '\xe8'+length
        print disasm(order,arch=arch)
        file.patch_address(srcaddr,[ord(i) for i in order])

binary = lief.parse("./vulner")
hook = lief.parse('./hook')

# write hook's .text content to binary's .eh_frame content
sec_ehrame = binary.get_section('.eh_frame')
print sec_ehrame.content
sec_text = hook.get_section('.text')
print sec_text.content
sec_ehrame.content = sec_text.content
print binary.get_section('.eh_frame').content

# hook target call
dstaddr = sec_ehrame.virtual_address
srcaddr = 0x400584

patch_call(binary,srcaddr,dstaddr)

binary.write('vulner.patched')
```

执行上面的脚本之后可以看到 patch 成功：

```
.eh_frame:0000000000400698 ; Segment type: Pure data
.eh_frame:0000000000400698 ; Segment permissions: Read
.eh_frame:0000000000400698 ; Segment alignment 'qword' can not be represented in assembly
.eh_frame:0000000000400698 _eh_frame       segment para public 'CONST' use64
.eh_frame:0000000000400698                 assume cs:_eh_frame
.eh_frame:0000000000400698                 ;org 400698h
.eh_frame:0000000000400698                 push    rbp
.eh_frame:0000000000400699                 mov     rbp, rsp
.eh_frame:000000000040069C                 mov     [rbp-8], rdi
.eh_frame:00000000004006A0                 mov     [rbp-0Ch], esi
.eh_frame:00000000004006A3                 mov     rsi, rdi
.eh_frame:00000000004006A6                 mov     rdi, 0
.eh_frame:00000000004006AD                 mov     rdx, 20h
.eh_frame:00000000004006B4                 mov     rax, 1
.eh_frame:00000000004006BB                 syscall                 ; LINUX - sys_write
.eh_frame:00000000004006BD                 nop
.eh_frame:00000000004006BE                 pop     rbp
.eh_frame:00000000004006BF                 retn
.eh_frame:00000000004006BF _eh_frame       ends
.eh_frame:00000000004006BF
LOAD:00000000004006C0 ; ===========================================================================
```

六、热补丁技术

热补丁是一种在程序运行时动态修补安全漏洞的技术，这种修补不需要重启操作系统或应用程序，因此能够大大增强系统的可用性。热补丁技术通常需要经过以下三个基本步骤：

首先，对程序中存在的安全漏洞进行详细的分析，明确漏洞成因，在此基础上编写相应的代码，并编译出可动态加载的补丁文件。

其次，通过加载程序将第一步得到的补丁文件加载到目标程序的内存空间，对于同一个系统，加载程序可以是通用的，补丁文件则因安全漏洞而异。

最后，修改程序的执行流程，把存在安全漏洞的代码替换为新的代码，完成热补丁的修补。

容易看出，热补丁技术的关键在于补丁文件的加载和程序执行流程的修改，工程上通常借助钩子技术（hook）来实现。钩子技术通过拦截系统调用、消息或事件，得到对系统进程或消息的控制权，进而改变或增强程序的行为。主流操作系统，如 Windows 和 Linux，都提供了 hook 的相应机制，并已被广泛运用到热补丁及代码调试等场景中。

1、Preload Hook

Preload Hook 是指利用操作系统对预加载（preload）的支持，将外部程序模块自动注入到指定的进程中的一种钩子技术。借助 Preload Hook，无需专门编写加载程序就能够实现补丁文件的加载，对于理解真正的热补丁技术有一定的帮助。

Preload Hook 有两种常见的用法：一种是配置环境变量 LD_PRELOAD，另一种是配置文件/etc/ld.so.preload。对于配置环境变量 LD_PRELOAD，通过命令行指定 LD_PRELOAD 将仅影响当前新进程及其子进程，写入全局环境变量则将影响所有新进程，但新进程的父进程可以控制子进程的环境变量从而取消 preload。配置文件/etc/ld.so.preload 则将对所有新进程生效且无法被取消。

示例如下。

初始程序源码：

original.c

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

int main()
{
    puts("A sample of preload hook.");
    sleep(2);
    puts("The end.");
    return 0;
}
```

补丁源码：

patch.c

```c
#include <stdio.h>

int sleep(int t)
{
    puts("Your sleep() is hook by xxx.");
}
```

补丁文件编译：

```
root@DESKTOP-C6VM2Q8:/home/project/pre# gcc -m32 -fPIC --shared patch.c -o patch.so
```

通过命令行进行 Preload Hook：

```
root@DESKTOP-C6VM2Q8:/home/project/pre# LD_PRELOAD=./patch.so ./original
A sample of preload hook.
Your sleep() is hook by xxx.
The end.
root@DESKTOP-C6VM2Q8:/home/project/pre#
```

可以看到已经成功地将系统的 sleep 函数修改成了自定义的功能。

2、热补丁的完整实现

利用 Preload Hook 虽然可以进行补丁修补，但它还不能算是真正的热补丁，因为对于已经处于运行状态的应用程序，这种方法是无法生效的。真正的热补丁必须通过专门的加载程序，利用动态的 hook 机制来实现补丁文件的加载和程序执行流程的修改。下面以 Linux 系统为例，介绍加载程序的编写及热补丁的完整实现。

初始程序源码：

original.c

```c
#include <stdio.h>
#include <unistd.h>
#include <time.h>

int main()
{
    while(1){
        sleep(5);
        printf("original: %ld\n", time(0));
    }
    return 0;
}
```

补丁源码：

patch.c

```c
#include <stdio.h>

int newprintf()
{
    puts("My student number is xxx.");
    return 0;
}
```

补丁文件编译：

```
root@DESKTOP-C6VM2Q8:/home/project/hotfix# gcc -m32 -fPIC --shared patch.c -o patch.so
```

Linux 系统提供了一种专门用于程序调试的系统调用 ptrace，热补丁的加载程序可以借助 ptrace 对运行状态的应用程序进行 hook，并最终实现热补丁修补。具体分为以下五个步骤。

第一步，加载程序通过 ptrace 关联（attach）到需要修补的进程上，并将该进程的寄存器及内存数据保存下来。代码如下。

```c
/* 关联到进程 */
void ptrace_attach(int pid)
{
    if(ptrace(PTRACE_ATTACH, pid, NULL, NULL) < 0) {
        perror("ptrace_attach");
        exit(-1);
    }

    waitpid(pid, NULL, /*WUNTRACED*/0);

    ptrace_readreg(pid, &oldregs);
}
```

第二步，得到指向 elf 文件的 link_map 链表的指针，并通过遍历 link_map 中的符号表，找到需要修补的 printf 函数及用于将补丁文件加载到进程内存地址空间的__libc_dlopen_mode 函数的地址。

根据 elf 文件的结构信息，首先从 elf 文件的开头开始读取信息，找到头部表(program header table),再根据头部表找到 elf 文件的全局偏移量表(global offset table,GOT 表)。GOT 表中的每一项都是一个 32bit 的 Elf32_Addr 地址，其中前两项是两个特殊的数据结构的地址：

GOT[0]为 PT_DYNAMIC 的起始地址

GOT[1]为 link_map 结构体的地址

由此可以得到指向 link_map 链表的指针。代码如下。

```c
/*
得到指向link_map链表首项的指针
*/
struct link_map *get_linkmap(int pid)
{
    Elf32_Ehdr *ehdr = (Elf32_Ehdr *) malloc(sizeof(Elf32_Ehdr));
    Elf32_Phdr *phdr = (Elf32_Phdr *) malloc(sizeof(Elf32_Phdr));
    Elf32_Dyn  *dyn =  (Elf32_Dyn *) malloc(sizeof(Elf32_Dyn));
    Elf32_Word got;
    struct link_map *map = (struct link_map *)malloc(sizeof(struct link_map));
    int i = 1;
    unsigned long tmpaddr;

    ptrace_read(pid, IMAGE_ADDR, ehdr, sizeof(Elf32_Ehdr));
    phdr_addr = IMAGE_ADDR + ehdr->e_phoff;
    printf("phdr_addr\t %p\n", phdr_addr);

    ptrace_read(pid, phdr_addr, phdr, sizeof(Elf32_Phdr));
    while(phdr->p_type != PT_DYNAMIC)
        ptrace_read(pid, phdr_addr += sizeof(Elf32_Phdr), phdr,sizeof(Elf32_Phdr));
    dyn_addr = phdr->p_vaddr;
    printf("dyn_addr\t %p\n", dyn_addr);

    ptrace_read(pid, dyn_addr, dyn, sizeof(Elf32_Dyn));
    while(dyn->d_tag != DT_PLTGOT) {
        tmpaddr = dyn_addr + i * sizeof(Elf32_Dyn);
        //printf("get_linkmap tmpaddr = %x\n",tmpaddr);
        ptrace_read(pid,tmpaddr, dyn, sizeof(Elf32_Dyn));
        i++;
    }

    got = (Elf32_Word)dyn->d_un.d_ptr;
    got += 4;
    //printf("GOT\t\t %p\n", got);

    ptrace_read(pid, got, &map_addr, 4);
    printf("map_addr\t %p\n", map_addr);
    map = map_addr;
    //ptrace_read(pid, map_addr, map, sizeof(struct link_map));

    free(ehdr);
    free(phdr);
    free(dyn);

    return map;
}
```

遍历 link_map 链表,依次对每一个 link_map 调用 find_symbol_in_linkmap 函数：

```c
/*
 解析指定符号
 */
unsigned long find_symbol(int pid, struct link_map *map, char *sym_name)
{
    struct link_map *lm = map;
    unsigned long sym_addr;
    char *str;
    unsigned long tmp;

    sym_addr = find_symbol_in_linkmap(pid, lm, sym_name);
    while(!sym_addr ) {
        ptrace_read(pid, (char *)lm+12, &tmp, 4);   //获取下一个库的link_map地址
        if(tmp == 0)
            return 0;
        lm = tmp;

        if ((sym_addr = find_symbol_in_linkmap(pid, lm, sym_name)))
            break;
    }

    return sym_addr;
}
```

find_symbol_in_linkmap 函数负责在指定的 link_map 中查找所需要的函数的地址：

```c
/*
在指定的link_map所指向的符号表中查找符号
*/
unsigned long find_symbol_in_linkmap(int pid, struct link_map *lm, char *sym_name)
{
    Elf32_Sym *sym = (Elf32_Sym *) malloc(sizeof(Elf32_Sym));
    int i = 0;
    char *str;
    unsigned long ret;
    int flags = 0;

    get_sym_info(pid, lm);

    do{
        if(ptrace_read(pid, symtab + i * sizeof(Elf32_Sym), sym, sizeof(Elf32_Sym)))
            return 0;
        i++;
        if (!sym->st_name && !sym->st_size && !sym->st_value)   //全为0是符号表的第一项
            continue;
        str = (char *) ptrace_readstr(pid, strtab + sym->st_name);
        if (strcmp(str, sym_name) == 0) {
            printf("\nfind_symbol_in_linkmap str = %s\n",str);
            printf("\nfind_symbol_in_linkmap sym->st_value = %x\n",sym->st_value);
            free(str);
            if(sym->st_value == 0)  //值为0代表这个符号本身就是重定向的内容
                continue;
            flags = 1;
            break;
        }
        free(str);
    }while(1);

    if (flags != 1)
        ret = 0;
    else
        ret =  link_addr + sym->st_value;

    free(sym);

    return ret;
}
```

第三步，调用__libc_dlopen_mode 函数，将补丁文件加载到需要修补的进程的内存空间中，并再一次遍历 link_map 中的符号表，找到新加载的补丁文件中的新函数 newprintf 的地址。代码如下。

```
/* 发现__libc_dlopen_mode，并调用它 */
sym_addr = find_symbol(pid, map, "__libc_dlopen_mode");        /* call _dl_open */
printf("found __libc_dlopen_mode at addr %p\n", sym_addr);
if(sym_addr == 0)
    goto detach;
call__libc_dlopen_mode(pid, sym_addr,libpath);    /* 注意装载的库地址 */
waitpid(pid,&status,0);
/* 找到新函数的地址 */
strcpy(sym_name, newfunname);                /* intercept */
sym_addr = find_symbol(pid, map, sym_name);
printf("%s addr\t %p\n", sym_name, sym_addr);
if(sym_addr == 0)
    goto detach;
```

第四步，找到要修补的函数 printf 的重定向地址，在该地址填入补丁文件中的新函数 newprintf 的地址。代码如下。

```
/* 找到旧函数在重定向表的地址 */
strcpy(sym_name, oldfunname);
rel_addr = find_sym_in_rel(pid, sym_name);
printf("%s rel addr\t %p\n", sym_name, rel_addr);
if(rel_addr == 0)
    goto detach;

/* 函数重定向 */
puts("intercept...");                    /* intercept */
if(modifyflag == 2)
    sym_addr = sym_addr - rel_addr - 4;
printf("main modify sym_addr = %x\n",sym_addr);
```

第五步，完成 patch，恢复现场，脱离需要修补的进程。代码如下。

```
    ptrace_write(pid, rel_addr, &sym_addr, sizeof(sym_addr));
    puts("patch ok");
detach:
    printf("prepare to detach\n");
    ptrace_detach(pid);

    return 0;
```

初始程序的执行效果：

```
root@DESKTOP-C6VM2Q8:/home/project/hotfix# ./original
original: 1622280172
original: 1622280177
```

通过加载程序打上热补丁后的执行效果：

```
root@DESKTOP-C6VM2Q8:/home/project/hotfix# ./original
original: 1622281125
original: 1622281130
original: 1622281135
original: 1622281140
patch successed
My student number is xxx.
My student number is xxx.
My student number is xxx.
My student number is xxx.
```

热补丁加载程序运行情况：

```
root@DESKTOP-C6VM2Q8:/home/project/hotfix# ./hotfix 11331 ./patch.so printf newprintf
main pid = 11331
main libpath : ./patch.so
main oldfunname : printf
main newfunname : newprintf
phdr_addr    0x8048034
dyn_addr     0x8049f14
map_addr     0xf7f82940

find_symbol_in_linkmap str = printf
find_symbol_in_linkmap sym->st_value = 50c60
found printf at addr 0xf7dbcc60
get_sym_info exit

find_symbol_in_linkmap str = __libc_dlopen_mode
find_symbol_in_linkmap sym->st_value = 131a90
found __libc_dlopen_mode at addr 0xf7e9da90
get_sym_info exit

find_symbol_in_linkmap str = newprintf
find_symbol_in_linkmap sym->st_value = 4bd
newprintf addr   0xf7f514bd
get_sym_info exit

printf rel addr  0x804a00c
intercept...
main modify sym_addr = f7f514bd
patch ok
prepare to detach
```

七、作业

1、参考实验手册的第三部分和第四部分，对二进制程序 overflow 进行修补。

程序 overflow 实现了一个非常简单的用户交互：输入学号，若输入的学号为 10 个字符，则在屏幕上打印一段感谢和表扬的话。程序共包含一个逻辑缺陷和一个栈溢出漏洞，要求同学们在没有源代码的情况下对其进行修补，修补后的程序仅打印与自己性别相对应的话，且无论输入多长的字符串均不会触发栈溢出漏洞。修改后的程序执行结果如下图所示：

```
Please input your student number:
a201900001
Thank you! You are a good girl.
```

2、参考实验手册的第五部分，对二进制程序 getshell 进行修补。

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。要求同学们利用 LIEF 库，将 getshell 程序中的 printf() 函数替换为补丁函数 newprint() ，获取系统的控制权，在当前目录下写入包含自己学号的文件，再将该文件显示出来。修改后的程序执行结果如下图所示：

```
# ./getshell
# whoami
root
# echo "a201900001" > num.txt
# cat num.txt
a201900001
# exit
```

八、分析报告

撰写分析报告，详细陈述完成作业的过程、方法及学习心得等，具体内容可参考实验手册的三至六部分。

九、扩展练习

学有余力的同学可以参考实验手册的第六部分，尝试用热补丁技术对二进制程序 repeat 进行修补。程序 repeat 会重复输出一句格言，将输出的内容修改为自己的姓名，修改后的程序执行结果如下图所示：

```
# ./repeat
Practice makes perfect.
Practice makes perfect.
Practice makes perfect.
Practice makes perfect.
Practice makes perfect.
patch successed
My name is zhangsan.
My name is zhangsan.
My name is zhangsan.
My name is zhangsan.
```