

**课 程 实 验 报 告**

**课程名称： 计算机系统基础**

**实验名称： ELF文件与程序链接**

**院 系 ：计算机科学与技术**

**专业班级 ： CS2306**

**学 号 ： U202315696**

**姓 名 ： 彭冲**

**指导教师 ： 朱虹**

**2025 年 4 月 17 日**

1. **实验目的与要求**

通过修改给定的可重定位的目标文件（链接炸弹），加深对可重定位目标文件格式、目标文件的生成、以及链接的理论知识的理解。

实验环境：Ubuntu

工具：GCC、GDB、readelf、hexdump、hexedit、od等。

1. **实验内容**

**任务** 链接炸弹的拆除

**在二进制层面，逐步修改构成目标程序“linkbomb”的多个二进制模块（“.o文件”），然后链接生成可执行程序，要求可执行程序运行能得到指定的效果。修改目标包括可重定位目标文件中的数据、机器指令、重定位记录等。**

### 1、第1关 数据节的修改

修改二进制可重定位目标文件 phase1.o 的数据节中的内容（不允许修改其**他节），使其与main.o链接后，生成的执行程序，可以输出自己的学号。**

### 2、第2关 简单的机器指令修改

修改二进制可重定位目标文件 phase2.o 的代码节中的内容（不允许修改其他节），使其与main.o链接后，生成的执行程序。在phase\_2.c 中，有一个静态函数 static void myfunc( ) ，要求在 do\_phase 函数中调用myfunc( )，显示信息myfunc is called. Good!。

### 3、第3关 有参数的函数调用的机器指令修改

修改二进制可重定位目标文件 phase3.o 的代码节中的内容（不允许修改其他节），使其与main.o链接后，生成的执行程序。在phase\_3.c 中，有一个静态函数 static void myfunc(int offset) ，要求在 do\_phase函数中调用myfunc(pos )，将do\_phase的参数pos直接传递myfunc，显示相应的信息。

### 4、第4关 有局部变量的机器指令修改

修改二进制可重定位目标文件 phase4.o 的代码节中的内容（不允许修改其他节），使其与main.o链接后，生成的执行程序。在phase\_4.c 中，有一个静态函数 static void myfunc(char \*s) ，要求在 do\_phase 函数中调用myfunc(s )，显示出自己的学号。

### 5、第5关 重定位表的修改

修改二进制可重定位目标文件 phase5.o 的重定位节中的内容（不允许修改代码节和数据节），使其与main.o链接后，生成的执行程序运行时，显示Class Name : Computer Foundation. Teacher Name : Xu Xiangyang。

### 6、第6关 强弱符号

不准修改 main.c 和phase6.o，通过增补一个文件，使得程序链接后，能够输出自己的学号。

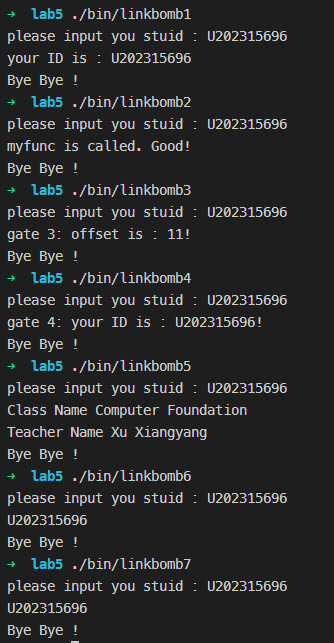
#gcc -no-pie -o linkbomb6 main.o phase6.o phase6\_patch.o

### 7、第7关 只读数据节的修改

修改 phase7.o 中只读数据节（不准修改代码节），使其与main.o链接后，能够输出自己的学号。

1. **实验记录及问题回答**

## （1）实验结果



图表 1 全部关的实验结果

## （2） 描述修改各个文件的基本思想及操作过程记录

1. **第1关 数据节的修改**

本关要求修改phase1.o的数据节中的内容,使程序输出自己的学号。

首先，使用objdump -d查看phase1.o的代码段内容，发现只有do\_phase一个函数，并使用readelf -r指令查看重定位表。

1. Disassembly of section .text:

2.

3. 0000000000000000 <do\_phase>:

4. 0: f3 0f 1e fa endbr64

5. 4: 55 push rbp

6. 5: 48 89 e5 mov rbp,rsp

7. 8: 48 83 ec 10 sub rsp,0x10

8. c: 89 7d fc mov DWORD PTR [rbp-0x4],edi

9. f: 8b 45 fc mov eax,DWORD PTR [rbp-0x4]

10. 12: 48 98 cdqe

11. 14: 48 8d 15 00 00 00 00 lea rdx,[rip+0x0] # 1b <do\_phase+0x1b>

12. 1b: 48 01 d0 add rax,rdx

13. 1e: 48 89 c6 mov rsi,rax

14. 21: 48 8d 3d 00 00 00 00 lea rdi,[rip+0x0] # 28 <do\_phase+0x28>

15. 28: b8 00 00 00 00 mov eax,0x0

16. 2d: e8 00 00 00 00 call 32 <do\_phase+0x32>

17. 32: 90 nop

18. 33: c9 leave

19. 34: c3 ret

Relocation section '.rela.text' at offset 0x300 contains 3 entries:

Offset Info Type Symbol's Value Symbol's Name + Addend

0000000000000017 0000000b00000002 R\_X86\_64\_PC32 0000000000000000 buf - 4

0000000000000024 0000000600000002 R\_X86\_64\_PC32 0000000000000000 .rodata - 4

000000000000002e 0000000f00000004 R\_X86\_64\_PLT32 0000000000000000 printf - 4

Relocation section '.rela.data.rel.local' at offset 0x348 contains 1 entry:

Offset Info Type Symbol's Value Symbol's Name + Addend

0000000000000000 0000000d00000001 R\_X86\_64\_64 0000000000000000 do\_phase + 0

Relocation section '.rela.eh\_frame' at offset 0x360 contains 1 entry:

Offset Info Type Symbol's Value Symbol's Name + Addend

0000000000000020 0000000200000002 R\_X86\_64\_PC32 0000000000000000 .text + 0

之后，结合重定位表中偏移Offset确定重定位位置，结合重定位类型如R\_X86\_64\_PC32可以确定重定位信息指向的内容。

do\_phase函数中关键逻辑：

11. 14: 48 8d 15 00 00 00 00 lea rdx, [buf]

12. 1b: 48 01 d0 add rax,rdx

13. 1e: 48 89 c6 mov rsi,rax

14. 21: 48 8d 3d 00 00 00 00 lea rdi, [.rodata]

15. 28: b8 00 00 00 00 mov eax,0x0

16. 2d: e8 00 00 00 00 call printf

x86-64平台下printf函数前6个参数通过rdi、rsi等寄存器传递，上面汇编程序中使用rdi、rsi寄存器传递了.rodata+0，buf+0处的参数。那么，下一步我们要查看这两处内容是什么。

使用readelf -r指令查看phase1.o中的符号表信息：

Symbol table '.symtab' contains 16 entries:

Num: Value Size Type Bind Vis Ndx Name

…

6: 0000000000000000 0 SECTION LOCAL DEFAULT 7 .rodata

…

11: 0000000000000000 40 OBJECT GLOBAL DEFAULT 3 buf

…

可知.rodata位于编号为7的节的0偏移处，buf位于编号为3的节的0偏移处。使用readelf -S指令查看节区编号等信息：

Section Headers:

[Nr] Name Type Address Off Size ES Flg Lk Inf Al

…

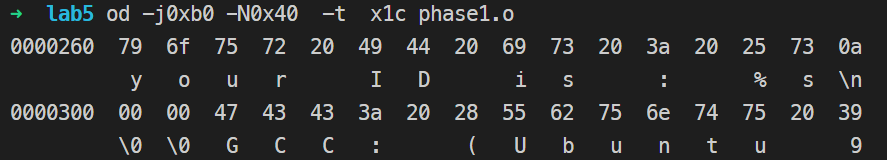
[ 3] .data PROGBITS 0000000000000000 000080 000028 00 WA 0 0 32

…

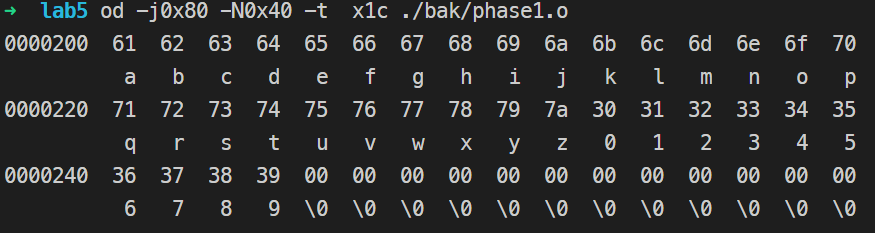
[ 7] .rodata PROGBITS 0000000000000000 0000b0 000011 00 A 0 0 1

结合上面节区信息中的文件内偏移（Off）信息，可知第一个参数位于.rodata只读数据节，文件内偏移为0xb0；第二个参数位于.data数据段，文件内偏移为0x80。

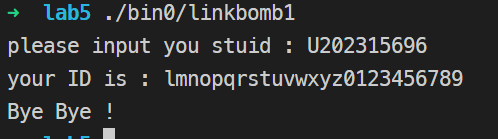
使用od -t x1c 指令查看第一个参数的内容，可知第一个参数为格式化字符串“your ID is : %s\n”。



由格式化字符串中的%s可知，第二个参数也为字符串，使用od -t x1 指令查看第二个参数的内容：

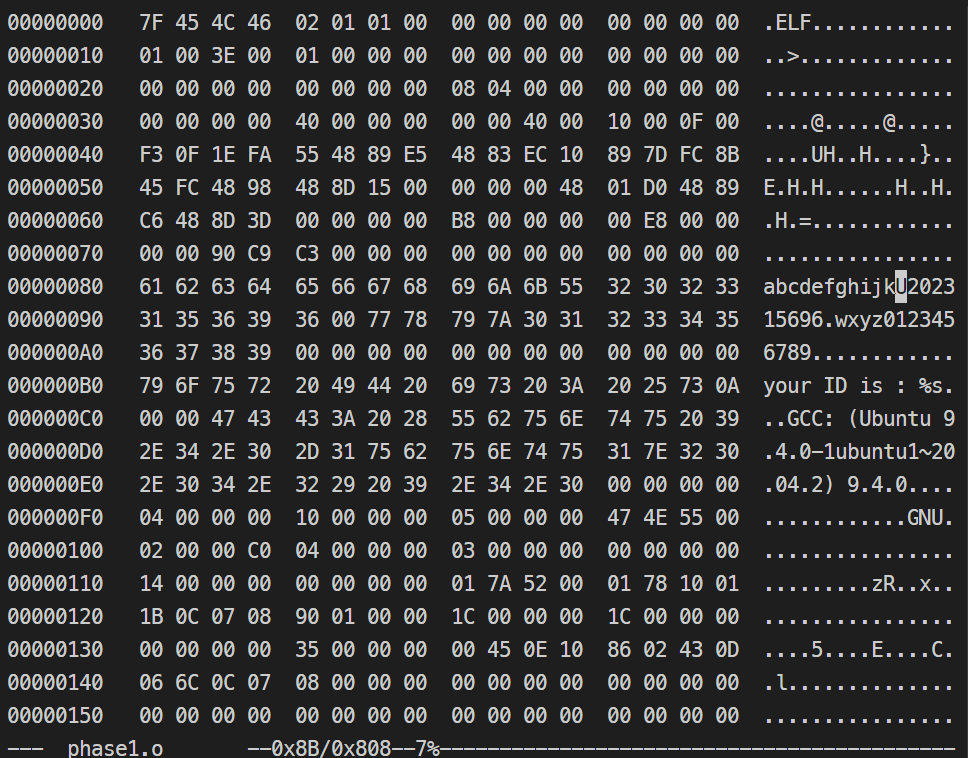


包含程序输出的字符串，位置差11，即传入do\_phase的cookie值。

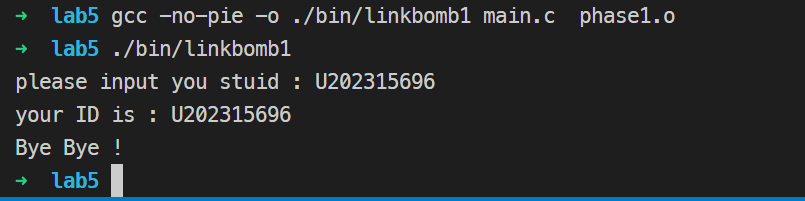


因此，只需将phase1.o中偏移为0x80+11=0x8b处填充学号字符串即可。

使用hexedit打开phase1.o文件，按F4键跳转到0x8b处，填充学号，并在末端填充一个00字节。



使用gcc重新编译，运行结果显示正确：



1. **第2关 简单的机器指令修改**

本关要求修改phase2.o的代码节中的内容，使程序运行myfunc函数。

首先，使用objdump -d查看phase2.o的代码段内容，发现有do\_phase和myfunc两个函数：

phase2.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <myfunc>:

0: f3 0f 1e fa endbr64

4: 55 push rbp

5: 48 89 e5 mov rbp,rsp

8: 48 8d 3d 00 00 00 00 lea rdi,[rip+0x0] # f <myfunc+0xf>

f: e8 00 00 00 00 call 14 <myfunc+0x14>

14: 90 nop

15: 5d pop rbp

16: c3 ret

0000000000000017 <do\_phase>:

17: f3 0f 1e fa endbr64

1b: 55 push rbp

1c: 48 89 e5 mov rbp,rsp

1f: 89 7d fc mov DWORD PTR [rbp-0x4],edi

22: 90 nop

23: 90 nop

…

34: 90 nop

35: 5d pop rbp

36: c3 ret

分析do\_phase函数，可见其没有调用myfunc函数，且在函数范围内存在大量nop空指令。因此，我们需要做的就是将nop指令修改为call指令调用myfunc函数。

汇编指令call的机器码为E8，其后跟4字节的偏移，值为call指令的下一条指令到myfunc首地址的偏移，即Offset = Addr(myfunc) - (rip)。由以上反汇编知，Addr(myfunc) = .text+0x0，(rip) = .text + 0x22+5 = .text + 0x27,故Offset = 0x0-0x27 = 0xFFFFFFD9。由小端法，即将.text段第0x22至0x27个字节改为E8 D9 FF FF FF。

使用readelf -S查看.text节的文件内偏移为0x40：

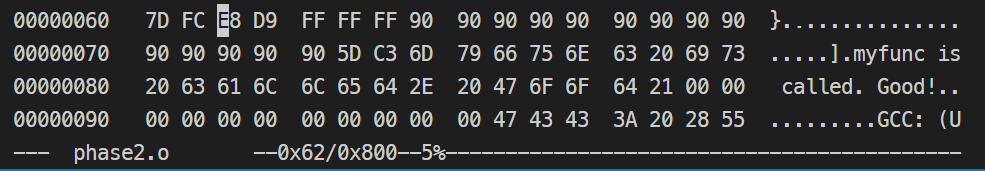
Section Headers:

[Nr] Name Type Address Off Size ES Flg Lk Inf Al

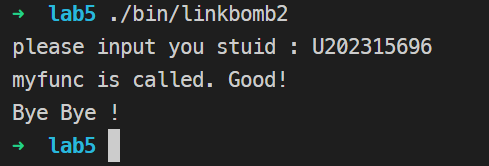
[ 0] NULL 0000000000000000 000000 000000 00 0 0 0

[ 1] .text PROGBITS 0000000000000000 000040 000037 00 AX 0 0 1

使用hexedit打开phase2.o，跳转到0x40+0x22 = 0x62处，修改5个字节为E8 D9 FF FF FF：



使用gcc重新链接，并运行，结果正确。



1. **第3关** 有参数的函数调用的机器指令修改

本关要求修改phase3.o的代码节中的内容，使程序运行myfunc函数。

首先，使用objdump -d查看phase3.o的代码段内容，可见有do\_phase和myfunc两个函数。

./phase3.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <myfunc>:

0: f3 0f 1e fa endbr64

4: 55 push rbp

5: 48 89 e5 mov rbp,rsp

8: 48 83 ec 10 sub rsp,0x10

c: 89 7d fc mov DWORD PTR [rbp-0x4],edi

f: 8b 45 fc mov eax,DWORD PTR [rbp-0x4]

12: 89 c6 mov esi,eax

14: 48 8d 3d 00 00 00 00 lea rdi,[rip+0x0] # 1b <myfunc+0x1b>

1b: b8 00 00 00 00 mov eax,0x0

20: e8 00 00 00 00 call 25 <myfunc+0x25>

25: 90 nop

26: c9 leave

27: c3 ret

0000000000000028 <do\_phase>:

28: f3 0f 1e fa endbr64

2c: 55 push rbp

2d: 48 89 e5 mov rbp,rsp

30: 89 7d fc mov DWORD PTR [rbp-0x4],edi

33: 90 nop

34: 90 nop

…

48: 90 nop

49: 5d pop rbp

4a: c3 ret

与上一关类似，同样需要将do\_phase中一些nop指令修改为call指令，并且，需要传递参数。

分析myfunc和do\_phase函数，二者均接受一个通过edi寄存器传递的参数。分析do\_phase函数指令，若想将参数传递给myfunc，需要将传入参数的值或内存中的地址赋给edi寄存器。那么，我们需要分析myfunc是怎样调用使用edi中的值的。

使用readelf -r指令查看重定位表，分析myfunc函数：

Relocation section '.rela.text' at offset 0x310 contains 2 entries:

Offset Info Type Symbol's Value Symbol's Name + Addend

0000000000000017 0000000500000002 R\_X86\_64\_PC32 0000000000000000 .rodata - 4

0000000000000021 0000000d00000004 R\_X86\_64\_PLT32 0000000000000000 printf – 4

可以得到myfunc中的符号信息：

0000000000000000 <myfunc>:

…

c: 89 7d fc mov DWORD PTR [rbp-0x4],edi

f: 8b 45 fc mov eax,DWORD PTR [rbp-0x4]

12: 89 c6 mov esi,eax

14: 48 8d 3d 00 00 00 00 lea rdi, [.rodata]

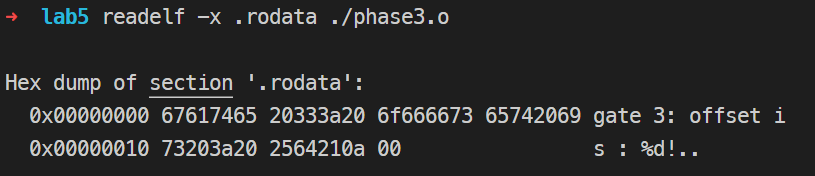
1b: b8 00 00 00 00 mov eax,0x0

20: e8 00 00 00 00 call printf

…

可知printf的两个参数依次为rdi=.rodata+0，esi=edi。

使用readelf -x查看.rodata节内容，为格式化字符串“gate 3: offset is : %d!\n”。由%d可知esi中应为do\_phase函数参数的值。



分析do\_phase函数的执行流程，发现不需要对其edi做任何修改，直接使用call指令即可继承edi寄存器中的值。

与上一关类似，Offset = Addr(myfunc) – (rip) = (.text+0x0)-(.text+0x33+5) = 0xFFFFFFC8，即将.text节第0x33至0x38的字节修改为E8 C8 FF FF FF。

使用readelf -S查看.text节的文件内偏移(Off)为0x40：

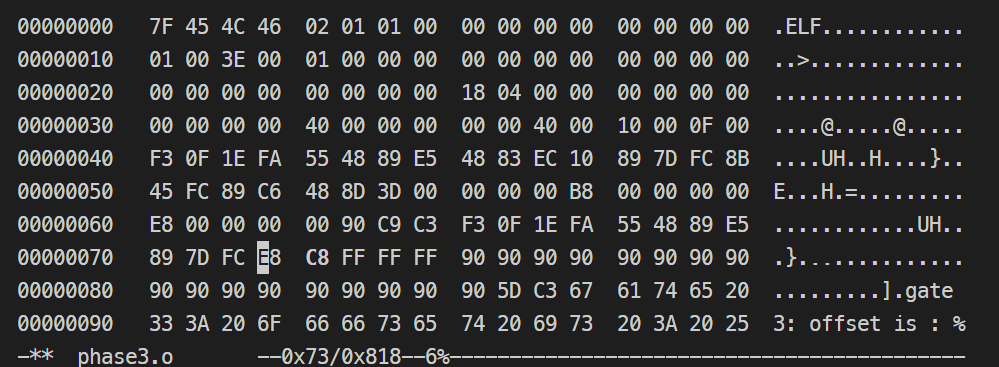
Section Headers:

[Nr] Name Type Address Off Size ES Flg Lk Inf Al

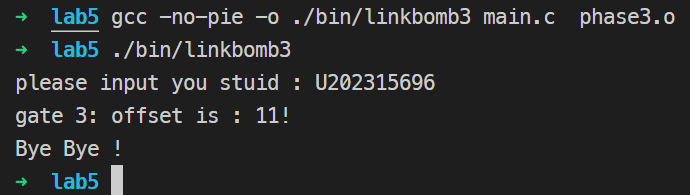
[ 0] NULL 0000000000000000 000000 000000 00 0 0 0

[ 1] .text PROGBITS 0000000000000000 000040 00004b 00 AX 0 0 1

使用hexedit打开phase3.o并跳转到0x40+0x33=0x73处，作出修改：



使用gcc重新链接生成，并运行，结果显示值为11。



分析源码中向do\_phase传入的参数，为gencookie函数根据学号生成的值，返回结果为学号最后一位数的数值加上5，即6+5=11，与运行结果符合。

1. **第4关** 有局部变量的机器指令修改

本关要求修改phase4.o的代码节中的内容，使程序运行myfunc函数。

首先，使用objdump -d查看phase4.o的代码段内容，可见有do\_phase和myfunc两个函数。

./phase4.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <myfunc>:

0: f3 0f 1e fa endbr64

4: 55 push rbp

5: 48 89 e5 mov rbp,rsp

8: 48 83 ec 10 sub rsp,0x10

c: 48 89 7d f8 mov QWORD PTR [rbp-0x8],rdi

10: 48 8b 45 f8 mov rax,QWORD PTR [rbp-0x8]

14: 48 89 c6 mov rsi,rax

17: 48 8d 3d 00 00 00 00 lea rdi,[rip+0x0] # 1e <myfunc+0x1e>

1e: b8 00 00 00 00 mov eax,0x0

23: e8 00 00 00 00 call 28 <myfunc+0x28>

28: 90 nop

29: c9 leave

2a: c3 ret

000000000000002b <do\_phase>:

2b: f3 0f 1e fa endbr64

2f: 55 push rbp

30: 48 89 e5 mov rbp,rsp

33: 48 83 ec 30 sub rsp,0x30

37: 89 7d dc mov DWORD PTR [rbp-0x24],edi

3a: 64 48 8b 04 25 28 00 mov rax,QWORD PTR fs:0x28

41: 00 00

43: 48 89 45 f8 mov QWORD PTR [rbp-0x8],rax

47: 31 c0 xor eax,eax

49: 48 b8 55 32 30 32 32 movabs rax,0x3332313232303255

50: 31 32 33

53: 48 89 45 ed mov QWORD PTR [rbp-0x13],rax

57: 66 c7 45 f5 34 35 mov WORD PTR [rbp-0xb],0x3534

5d: c6 45 f7 00 mov BYTE PTR [rbp-0x9],0x0

61: 90 nop

62: 90 nop

…

74: 90 nop

75: 48 8b 45 f8 mov rax,QWORD PTR [rbp-0x8]

79: 64 48 33 04 25 28 00 xor rax,QWORD PTR fs:0x28

80: 00 00

82: 74 05 je 89 <do\_phase+0x5e>

84: e8 00 00 00 00 call 89 <do\_phase+0x5e>

89: c9 leave

8a: c3 ret

与上一关类似，同样需要将do\_phase中一些nop指令修改为call指令，并且，需要传递参数。

分析do\_phase函数，其中以下几行指令，是将0x3332313232303255、0x3534、0x0存入栈区，即rbp-0x13至rbp-0x911个字节为55 32 30 32 32 31 32 33 34 35 00，解释成字符串为“U202212345”。

49: 48 b8 55 32 30 32 32 movabs rax,0x3332313232303255

50: 31 32 33

53: 48 89 45 ed mov QWORD PTR [rbp-0x13],rax

57: 66 c7 45 f5 34 35 mov WORD PTR [rbp-0xb],0x3534

5d: c6 45 f7 00 mov BYTE PTR [rbp-0x9],0x0

因此，首先，我们可以将以上数字修改成自己的学号对应的整数，然后，设法将其传给myfunc函数。

1. 修改学号

由以上反汇编代码可得两个整数的地址为.text+0x4b和.text+0x5b。使用readelf -S指令，查看.text节的文件内偏移Off为0x40。

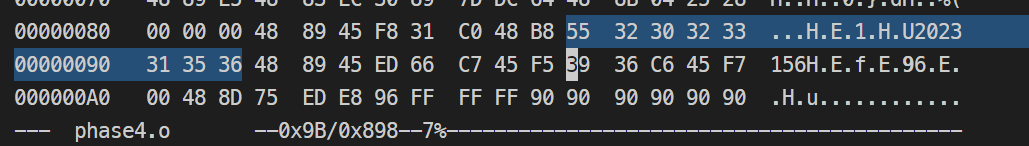
Section Headers:

[Nr] Name Type Address Off Size ES Flg Lk Inf Al

[ 0] NULL 0000000000000000 000000 000000 00 0 0 0

[ 1] .text PROGBITS 0000000000000000 000040 00008b 00 AX 0 0 1

使用hexedit打开phase4.o，分别跳转到0x8b,0x9b处修改字节为学号：



1. 调用myfunc

编写汇编指令，将栈区内的字节起始地址传给rdi，然后调用myfunc。

如何计算myfunc的偏移？首先只编译一行指令，得到该指令占4个字节，故偏移为0x0-0x61-4-5=-0x6a=0FFFFFF96

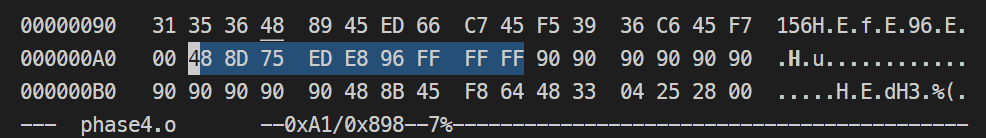
.intel\_syntax noprefix

lea rdi, [rbp-0x13]

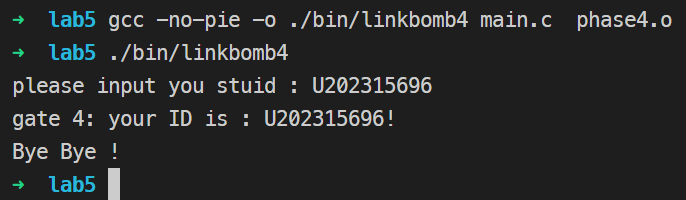
call -0x6a

使用as指令将其编译汇编，可得到对应机器码48 8d 7d ed e8 96 ff ff ff。

使用hexedit打开phase4.o，并跳转到0x40+0x61(do\_phase函数第一个nop的位置偏移) = 0xa1处，作出修改：



完成以上两步后，使用gcc重新链接生成，运行结果正确：



1. **第5关** 重定位表的修改

本关要求修改phase5.o的重定位节中的内容，使程序运行显示Class Name : Computer Foundation. Teacher Name : Xu Xiangyang。

首先，使用objdump -d查看phase5.o的代码段内容，可见有do\_phase函数。再使用readelf -r指令查看重定位表。

./phase5.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <do\_phase>:

0: f3 0f 1e fa endbr64

4: 55 push rbp

5: 48 89 e5 mov rbp,rsp

8: 48 83 ec 10 sub rsp,0x10

c: 89 7d fc mov DWORD PTR [rbp-0x4],edi

f: 48 8d 35 00 00 00 00 lea rsi,[rip+0x0] # 16 <do\_phase+0x16>

16: 48 8d 3d 00 00 00 00 lea rdi,[rip+0x0] # 1d <do\_phase+0x1d>

1d: b8 00 00 00 00 mov eax,0x0

22: e8 00 00 00 00 call 27 <do\_phase+0x27>

27: 48 8d 35 00 00 00 00 lea rsi,[rip+0x0] # 2e <do\_phase+0x2e>

2e: 48 8d 3d 00 00 00 00 lea rdi,[rip+0x0] # 35 <do\_phase+0x35>

35: b8 00 00 00 00 mov eax,0x0

3a: e8 00 00 00 00 call 3f <do\_phase+0x3f>

3f: 90 nop

40: 90 nop

…

54: 90 nop

55: c9 leave

56: c3 ret

Relocation section '.rela.text' at offset 0x3f8 contains 6 entries:

Offset Info Type Symbol's Value Symbol's Name + Addend

0000000000000012 0000000d00000002 R\_X86\_64\_PC32 0000000000000040 originalclass - 4

0000000000000019 0000000600000002 R\_X86\_64\_PC32 0000000000000000 .rodata - 4

0000000000000023 0000001200000004 R\_X86\_64\_PLT32 0000000000000000 printf - 4

000000000000002a 0000000e00000002 R\_X86\_64\_PC32 0000000000000060 originalteacher - 4

0000000000000031 0000000600000002 R\_X86\_64\_PC32 0000000000000000 .rodata + b

000000000000003b 0000001200000004 R\_X86\_64\_PLT32 0000000000000000 printf - 4

Relocation section '.rela.data.rel.local' at offset 0x488 contains 1 entry:

...

结合重定位表信息，可以分析do\_phase函数主要内容：

f: 48 8d 35 00 00 00 00 lea rsi,[originalclass]

16: 48 8d 3d 00 00 00 00 lea rdi,[.rodata+0]

1d: b8 00 00 00 00 mov eax,0x0

22: e8 00 00 00 00 call printf

27: 48 8d 35 00 00 00 00 lea rsi,[originalteacher]

2e: 48 8d 3d 00 00 00 00 lea rdi,[.rodata + b + 4] # 35 <do\_phase+0x35>

35: b8 00 00 00 00 mov eax,0x0

3a: e8 00 00 00 00 call printf

使用readelf -s指令查看符号具体信息：

Symbol table '.symtab' contains 19 entries:

Num: Value Size Type Bind Vis Ndx Name

...

13: 0000000000000040 20 OBJECT GLOBAL DEFAULT 3 originalclass

14: 0000000000000060 20 OBJECT GLOBAL DEFAULT 3 originalteacher

可见originalclass与originalteacher均位于编号为3的节的0x40和0x60偏移处。

使用readelf -S指令查看节头信息：

Section Headers:

[Nr] Name Type Address Off Size ES Flg Lk Inf Al

...

[ 2] .rela.text        RELA            0000000000000000 0003f8 000090 18   I 13   1  8

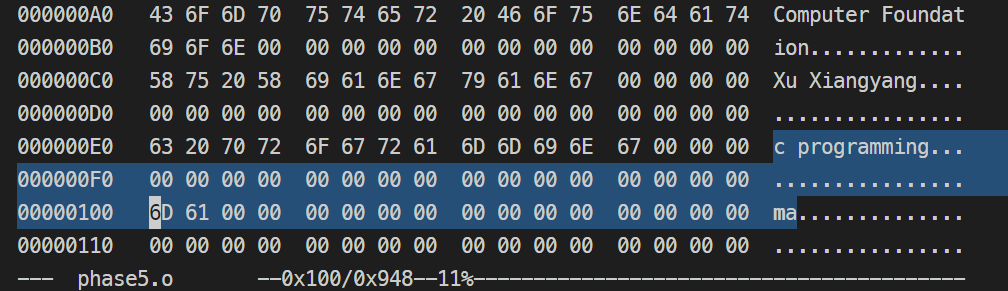
[ 3] .data PROGBITS 0000000000000000 0000a0 000074 00 WA 0 0 16

...

[ 7] .rodata PROGBITS 0000000000000000 000120 000020 00 A 0 0 1

可见编号为3的节为.data节，文件内偏移为0xa0，故originalclass位于0xa0+0x40=0xe0，originalteacher位于0xa0+0x60=0x100处。

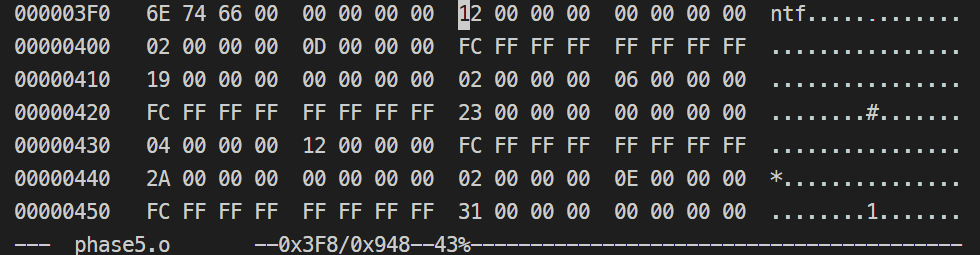
使用hexedit查看这两处内容，为“c programming”和“ma”，即当前输出的课程名和教师名。而“Computer Fundation”和“Xu Xiangyang”分别位于0xa0和0xc0处，即各自位置减去0x40处。



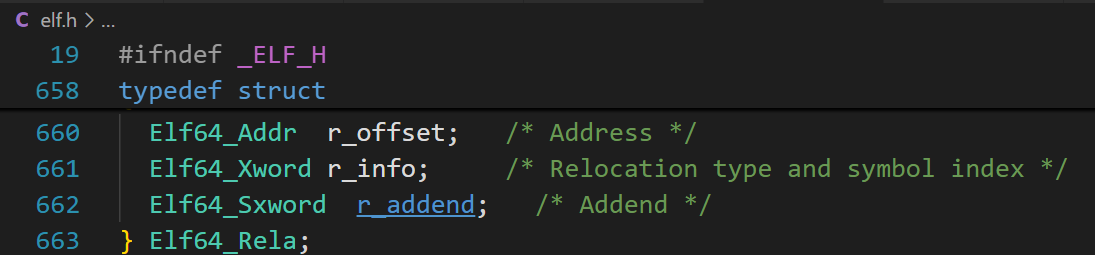
因此，可以修改重定位节中originalclass与originalteacher两个符号的Addend值，将各自的值减去0x40，即可将两个符号指向关卡要求的字符串。

由前面readelf -S指令结果知.rela.text节的文件内偏移为0x3f8，且ES即entry size大小为0x18

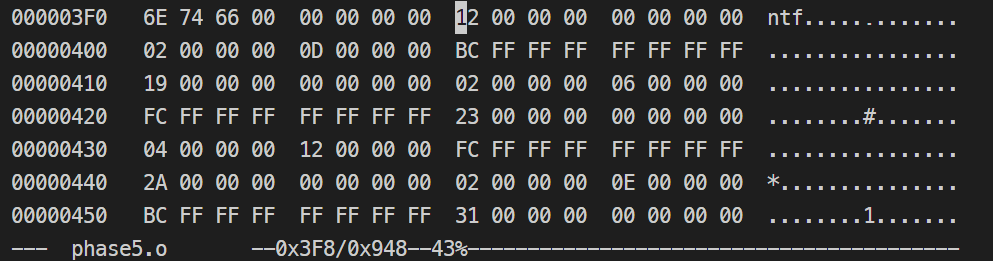
，重定位节中originalclass与originalteacher两个符号编号分别为0和3，故两个符号的重定位信息位于0x3f8+0\*0x18=0x3f8和0x3f8+3\*0x18=0x448。



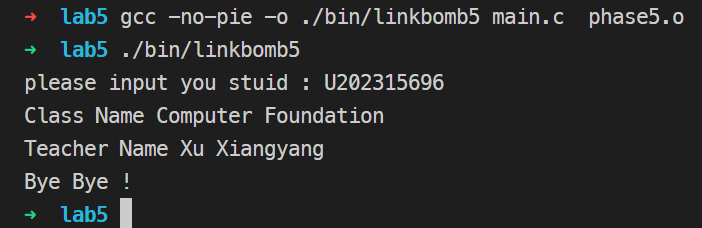
由/usr/include/elf.h中数据结构Elf64\_Rela的定义知addend值位于最后8个字节；



故将0x38f+0x10=0x408和0x448+0x10=0x458处的字节改成0xFFFFFFBC(-0x44=-0x4-0x40)



使用gcc重新链接生成，运行结果正确。



1. **第6关** 强弱符号

本关要求增补一个文件，使得程序链接后，能够输出自己的学号。

首先，使用objdump -d查看phase6.o的代码段内容，可见有do\_phase函数。再使用readelf -r指令查看重定位表。

./phase6.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <do\_phase>:

0: f3 0f 1e fa endbr64

4: 55 push rbp

5: 48 89 e5 mov rbp,rsp

8: 48 83 ec 10 sub rsp,0x10

c: 89 7d fc mov DWORD PTR [rbp-0x4],edi

f: 48 8b 05 00 00 00 00 mov rax,QWORD PTR [rip+0x0] # 16 <do\_phase+0x16>

16: 48 85 c0 test rax,rax

19: 74 10 je 2b <do\_phase+0x2b>

1b: 48 8b 15 00 00 00 00 mov rdx,QWORD PTR [rip+0x0] # 22 <do\_phase+0x22>

22: b8 00 00 00 00 mov eax,0x0

27: ff d2 call rdx

29: eb 0c jmp 37 <do\_phase+0x37>

2b: 48 8d 3d 00 00 00 00 lea rdi,[rip+0x0] # 32 <do\_phase+0x32>

32: e8 00 00 00 00 call 37 <do\_phase+0x37>

37: 90 nop

38: c9 leave

39: c3 ret

Relocation section '.rela.text' at offset 0x2e8 contains 4 entries:

Offset Info Type Symbol's Value Symbol's Name + Addend

0000000000000012 0000000d00000002 R\_X86\_64\_PC32 0000000000000008 myprint - 4

000000000000001e 0000000d00000002 R\_X86\_64\_PC32 0000000000000008 myprint - 4

000000000000002e 0000000600000002 R\_X86\_64\_PC32 0000000000000000 .rodata - 4

0000000000000033 0000000f00000004 R\_X86\_64\_PLT32 0000000000000000 puts - 4

结合重定位表分析do\_phase函数，发现关键在于以下

1b: 48 8b 15 00 00 00 00 mov rdx, QWORD PTR [myprint]

22: b8 00 00 00 00 mov eax,0x0

27: ff d2 call rdx

即调用跳转到myprint指向的地址。

使用readelf -s查看myprint的符号信息，发现其位于COM节：

Symbol table '.symtab' contains 16 entries:

Num: Value Size Type Bind Vis Ndx Name

...

13: 0000000000000008 8 OBJECT GLOBAL DEFAULT COM myprint

查阅资料，COM即COMMON，用于标记未初始化的全局变量或静态变量或extern函数。本例中，myprint为大小8字节的GLOBAL OBJECT，故为未初始化的全局变量，为**弱符号**。

因此，在增补文件中定义并初始化myprint全局变量，使之为强符号，并使其值指向一个函数，这个函数用来打印学号。编写linux汇编代码：

section .data

stuid db "U202315696", 10, 0

align 8

myprint dq 0

section .text

extern puts

global myprint

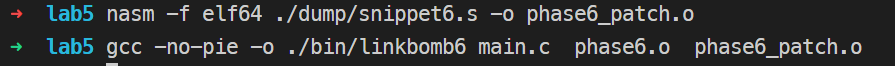
myfunc:

mov rdi, stuid

call puts

ret

使用nasm命令生成目标文件，并使用gcc命令链接：



此时还差最后一步，找到myfunc函数在链接后的虚拟地址。可以使用符号表查看，也可以直接使用objdump命令反汇编查看。可得到虚拟地址为0x401360。

因此，将上面汇编代码中myprint的值初始化为0x401360即可，即：

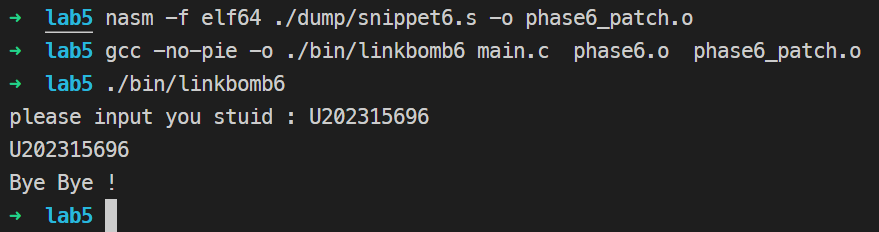
section .data

stuid db "U202315696", 0

align 8

myprint dq 0x401360

使用nasm命令生成目标文件，并使用gcc命令链接，运行结果正确：



1. **第7关 只读数据节的修改**

本关与第一关几乎相同，区别只有一个修改数据节，一个修改只读数据节。

首先，使用objdump -d查看phase7.o的代码段内容，发现只有do\_phase一个函数，并使用readelf -r指令查看重定位表。

./phase7.o: file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <do\_phase>:

0: f3 0f 1e fa endbr64

4: 55 push rbp

5: 48 89 e5 mov rbp,rsp

8: 48 83 ec 10 sub rsp,0x10

c: 89 7d fc mov DWORD PTR [rbp-0x4],edi

f: 48 8d 3d 00 00 00 00 lea rdi,[rip+0x0] # 16 <do\_phase+0x16>

16: e8 00 00 00 00 call 1b <do\_phase+0x1b>

1b: 90 nop

1c: c9 leave

1d: c3 ret

Relocation section '.rela.text' at offset 0x298 contains 2 entries:

Offset Info Type Symbol's Value Symbol's Name + Addend

0000000000000012 0000000600000002 R\_X86\_64\_PC32 0000000000000000 .rodata - 4

0000000000000017 0000000e00000004 R\_X86\_64\_PLT32 0000000000000000 puts - 4

  ...

结合重定位表可知是调用puts函数输出.rodata+0处的字符串。

使用readelf -S指令查看.rodata节信息，可得文件内偏移为0x68：

Section Headers:

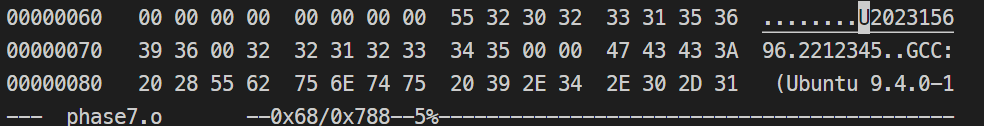
[Nr] Name Type Address Off Size ES Flg Lk Inf Al

...

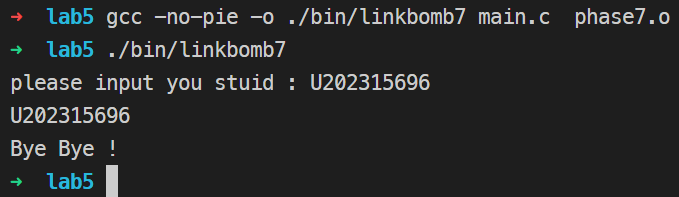
[ 7] .rodata PROGBITS 0000000000000000 000068 000013 00 A 0 0 1

...

使用hexedit修改0x68处字节为学号即可。



使用gcc重新链接生成，运行结果正确：



1. **体会**

本次实验