哈爾濱工業大學

实验报告

实 验（五）

题　　目 LinkLab

链接

专　　业 计算机类

学　　号 1190200910

班　　级 1903012

学　　生 严幸

指 导 教 师 史先俊

实 验 地 点 G709

实 验 日 期 2021-05-19

计算机科学与技术学院

目 录

第 1 章 实验基本信息

1.1 实验目的

- 理解链接的作用与工作步骤
- 掌握 ELF 结构与符号解析与重定位的工作过程
- 熟练使用 Linux 工具完成 ELF 分析与修改

1.2 实验环境与工具

1.2.1 硬件环境

X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位；

1.2.3 开发工具

Visual Studio 2010 64 位以上；GDB/OBJDUMP；DDD/EDB 等

1.3 实验预习

上实验课前，必须认真预习实验指导书（PPT 或 PDF）

了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。

请按顺序写出 ELF 格式的可执行目标文件的各类信息

请按照内存地址从低到高的顺序，写出 Linux 下 X64 内存映像。

请运行“LinkAddress -u 学号 姓名” 按地址循序写出各符号的名称、地址、空间。并按照 Linux 下 X64 内存映像标出其所属各区。 (为方便统一，请用 gcc –no-pie –fno-PIC 编译与连接)

请按顺序写出 LinkAddress 从开始执行到 main 前/后执行的子程序的名字或地址。(gcc 与 objdump/GDB/EDB)

第 2 章 实验预习

2.1 请按顺序写出 ELF 格式的可执行目标文件的各类信息（5 分）

ELF 文件由 4 部分组成，分别是 ELF 头（ELF header）、程序头表（Program header table）、节（Section）和节头表（Section header table）。[7]

(1) ELF 头（ELF header）

ELF 头是一个结构体，包含这些属性：e_ident, e_type, e_machine, e_version, e_entry, e_phoff, e_shoff, e_flags, e_ehsize, e_phentsize, e_phnum, e_shentsize, e_shnum, e_shstrndx。

ELF 最开头是 16 个字节的 e_ident, 其中包含用以表示 ELF 文件的字符，以及其他一些与机器无关的信息。开头的 4 个字节值固定不变，为 0x7f 和 ELF 三个字符。其他属性的含义如下：

e_type 它标识的是该文件的类型。
e_machine 表明运行该程序需要的体系结构。
e_version 表示文件的版本。
e_entry 程序的入口地址。
e_phoff 表示 Program header table 在文件中的偏移量（以字节计数）。
e_shoff 表示 Section header table 在文件中的偏移量（以字节计数）。
e_flags 对 IA32 而言，此项为 0。
e_ehsize 表示 ELF header 大小（以字节计数）。
e_phentsize 表示 Program header table 中每一个条目的大小。
e_phnum 表示 Program header table 中有多少个条目。
e_shentsize 表示 Section header table 中的每一个条目的大小。
e_shnum 表示 Section header table 中有多少个条目。
e_shstrndx 包含节名称的字符串是第几个节（从零开始计数）。

(2) 程序头表（Program header table）

程序头表也是一个结构体，它描述的是一个段在文件中的位置、大小以及它被放进内存后所在的位置和大小。每个属性及其含义如下：

p_type 当前 Program header 所描述的段的类型。
p_offset 段的第一个字节在文件中的偏移。
p_vaddr 段的一个字节在内存中的虚拟地址
p_paddr 在物理内存定位相关的系统中，此项是为物理地址保留。
p_filesz 段在文件中的长度。

p_memsz 段在内存中的长度。
p_flags 与段相关的标志。
p_align 根据此项值来确定段在文件及内存中如何对齐。

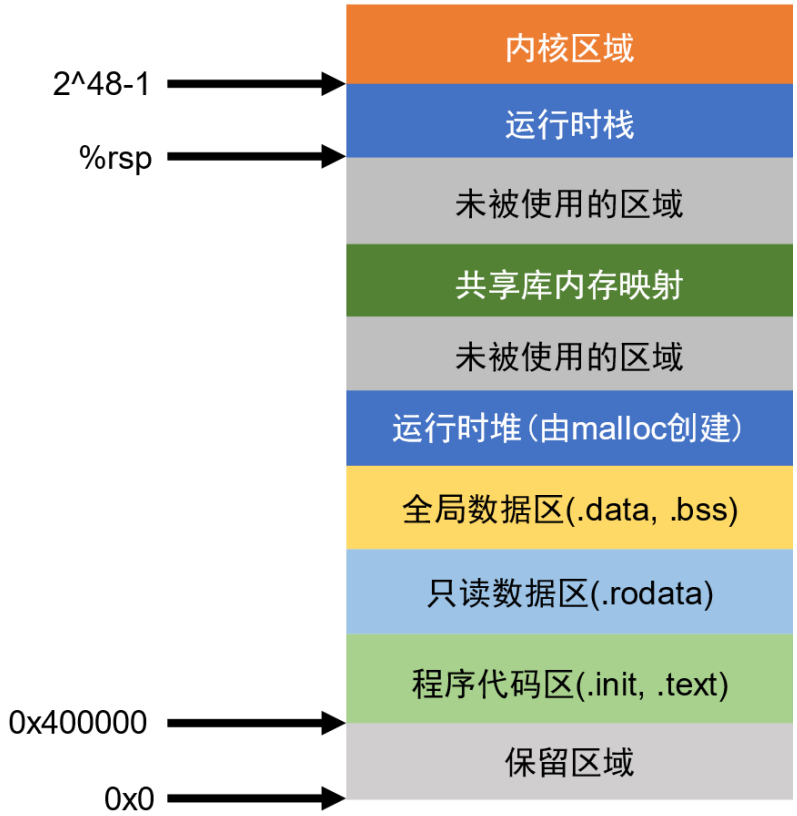
(3) 节（Section）

ELF 文件包含诸多节。举例如下：
.bss：放置未初始化或初始化为 0 的全局变量
.data：防止初始化不为 0 的全局变量
.rodata：只读数据区，例如字符串常量
.init：包含进程初始化时要执行的指令，当程序开始运行时，系统会在进入主函数之前执行这一段代码
.text：程序代码
.symtab：存放符号表
.debug：调试信息，内容没有统一规定
.line：用于调试，包含那些调试符号的行号
.strtab：存放字符串表
.plt：包含函数链接表
.got：包含全局偏移表

(4) 节头表（Section header table）

节头标结构体包含这些属性：
sh_name; 表示节的名字
sh_type; 节的类型
sh_flags; 成员 sh_flags 用来描述该节的诸多属性。它由一系列标志比特位组成，每一位对应一个属性。
sh_addr; 如果该节的数据内容最后存在于程序的内存映像中，成员 sh_addr 指定的是该节数据在内存中的起始地址。否则，即该节数据不需要映射到内存中时，该成员数值为 0。
sh_offset; 成员 sh_offset 的数值代表的意思是该节数据在目标文件中的偏移量。需要注意的是，当节的类型为 SHT_NOBITS 时（例如.bss 节），说明该节在目标文件中并不占用空间大小，所以此时成员 sh_offset 的数值代表的是概念上的偏移值。
sh_size; 成员 sh_size 说明的是该节的大小。
sh_link; 成员 sh_link 的数值代表的是一个节头表索引值
sh_info; 成员 sh_info 的意义会因节的类型的不同而不同。
sh_addralign; 有一些节有字节对齐的限制。
sh_entsize; 有一些节的数据内容是固定大小的表，例如符号表。

2.2 请按照内存地址从低到高的顺序，写出 Linux 下 X64 内存映像。（5 分）



2.3 请运行“LinkAddress -u 学号 姓名” 按地址循序写出各符号的地址、空间。并按照 Linux 下 X64 内存映像标出其所属各区。

（5 分）

我使用的编译链接方式是：
gcc -Og -no-pie -fno-PIC -fno-omit-frame-pointer -fno-stack-protector -fcf-protection=none -mmanual-endbr -g
这样会使得共享函数与 main 在同一段。

符号名	类型	地址	空间	所属区
free	函数名	0x401030	0x10Byte	代码区
strcpy	函数名	0x401040	0x20Byte	代码区
printf	函数名	0x401060	0x10Byte	代码区
malloc	函数名	0x401070	0x20Byte	代码区
exit	函数名	0x401090	0xF6Byte	代码区
useless	函数名	0x401186	0x6Byte	代码区
show_pointer	函数名	0x40118c	0x23Byte	代码区
main	函数名	0x4011af	未知	代码区
local pstr	char *	0x402170	112Byte	只读数据区
pstr	char *	0x4021e8	40Byte	只读数据区

cc	const char[100]	0x402220	100Byte	只读数据区
gc	const int	0x402284	4Byte	只读数据区
cstr	char[100]	0x4040a0	100Byte	.data 数据区
glong	long	0x404108	8Byte	.data 数据区
global	int	0x404110	4Byte	.data 数据区
gint0	int	0x40412c	4Byte	.data 数据区
huge array	char[1L<<30]	0x404140	1 GB	.bss 数据区
big array	char[1L<<24]	0x40404140	16 MB	.bss 数据区
p5	void *	0x7fcfb2282010	2GB	运行时堆
p4	void *	0x7fd032283010	1GB	运行时堆
p3	void *	0x7fd072284010	128KB+1B	运行时堆
p2	void *	0x7fd0722a5010	128KB	运行时堆
p1	void *	0x7fd0722c6010	256MB	运行时堆
argc	int	0x7ffda90de96c	4Byte	运行时栈
local astr	char[1001]	0x7ffda90de970	1001Byte	运行时栈
local int 0	int	0x7ffda90ded6c	4Byte	运行时栈
local int 1	int	0x7ffda90ded68	4Byte	运行时栈
argv	char *[]	0x7ffda90dee98	4Byte	运行时栈
env	char**	0x7ffda90deec0	8Byte	运行时栈
argv[0]	char*	0x7ffda90e11c0	4Byte	运行时栈
argv[1]	char*	0x7ffda90e11c5	2Byte	运行时栈
argv[2]	char*	0x7ffda90e11c8	10Byte	运行时栈
argv[3]	char*	0x7ffda90e11d3	7Byte	运行时栈
env[0]	char*	0x7ffda90e11db	字符串长度	运行时栈
其余 env[…]略去				

2.4 请按顺序写出 LinkAddress 从开始执行到 main 前/后执行的子程序的名字。（gcc 与 objdump/GDB/EDB）（5 分）

调用 main 函数之前：

_dl_start

_dl_setup_hash

_dl_sysdep_start

_dl_init

_init

_dl_vdso_vsym

_dl_lookup_symbol_x

do_lookup_x

_dl_name_match_p

__init_misc
@plt
__ctype_init
check_stdfiles_vtables
init_cacheinfo
handle_intel.constprop.1
intel_check_word.isra
_start
_libc_start_main
__cxa_atexit
__new_exitfn
_libc_csu_init
_init
frame_dummy
register_tm_clones
_setjmp
__sigsetjmp
__sigjmp_save

调用 main 函数之后：
exit
__run_exit_handlers
__call_tls_dtors
dl_fini
rtld_lock_default_lock_recursive
_dl_sort_maps
memset
rtld_lock_default_unlock_recursive
__do_global_dtors_aux
deregister_tm_clones
_dl_fini
_IO_cleanup
_IO_flush_all_lockp
_exit

第 3 章 各阶段的原理与方法

每阶段 40 分，phasex.o 20 分，分析 20 分，总分不超过 80 分

3.1 阶段 1 的分析

程序运行结果截图：

```
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 ls
main.o  phase1.o  phase1.out  phase1.s  phase2.o  phase3.o  phase4.o  phase5.o
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 gcc -no-pie -o phase1_result m
ain.o phase1.o
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 ./phase1_result
1190200910
```

分析与设计的过程：

首先直接编译链接 main.o 和 phase1.o，并运行，可以看到输出了乱码。

```
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 gcc -no-pie -o phase1.out main
.o phase1.o
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 ls
linkbomb  main.o  phase1.o  phase1.out  phase1.s  phase2.o  phase3.o  phase4.o  phase5.o
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 ./phase1.out
Vy4szjlQACb68YoR04H kMbmU9U65kvgeH1u9RoxTzOHF16FG13XEVj2 0sPlsweN      cFsr4GcrRKCZQwiYFlyHdAz
 yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910
```

使用 objdump 看一下链接 phase1.o 后生成的可执行文件的 do_phase 函数:

```
0000000000401176 <do_phase>:
  401176:       f3 0f 1e fa             endbr64
  40117a:       55                      push   %rbp
  40117b:       48 89 e5                mov    %rsp,%rbp
  40117e:       bf 40 40 40 00          mov    $0x404040,%edi
  401183:       e8 b8 fe ff ff          callq  401040 <puts@plt>
  401188:       90                      nop
  401189:       5d                      pop    %rbp
  40118a:       c3                      retq
  40118b:       0f 1f 44 00 00          nopl   0x0(%rax,%rax,1)
```

在 callq puts 函数之前，将 0x404040 作为 puts 的第一个参数传到了 edi 寄存器，说明这个地方应该存着字符串常量的地址，应该在.data 节。

然后使用 readelf -S 看一眼 phase1.o 文件的节头信息：

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 readelf -S phase1.o
There are 14 section headers, starting at offset 0x388:

节头：
  [号] 名称              类型             地址              偏移量
       大小              全体大小          旗标    链接    信息    对齐
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       0000000000000015  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  000002b0
       0000000000000030  0000000000000018   I      11     1     8
  [ 3] .data             PROGBITS         0000000000000000  00000060
       0000000000000068  0000000000000000  WA       0     0     32
  [ 4] .rela.data        RELA             0000000000000000  000002e0
       0000000000000018  0000000000000018   I      11     3     8
  [ 5] .bss              NOBITS           0000000000000000  000000c8
       0000000000000000  0000000000000000  WA       0     0     1
  [ 6] .comment          PROGBITS         0000000000000000  000000c8
       000000000000002d  0000000000000001  MS       0     0     1
```

其中我们需要重点关注.data 节在文件中的偏移，这里是偏移 0x60 个字节。因此马上修改 ELF 文件中.data 节的内容时，就从文件的第 0x60 个字节开始修改。

然后我们用 objdump -d 看一眼 phase1.o 的反汇编代码：

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 objdump -d phase1.o

phase1.o:     文件格式 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    %rsp,%rbp
   8:   bf 00 00 00 00          mov    $0x0,%edi
   d:   e8 00 00 00 00          callq  12 <do_phase+0x12>
  12:   90                      nop
  13:   5d                      pop    %rbp
  14:   c3                      retq
```

上图中的用红框框起来的部分就是需要重定位所在的地址。

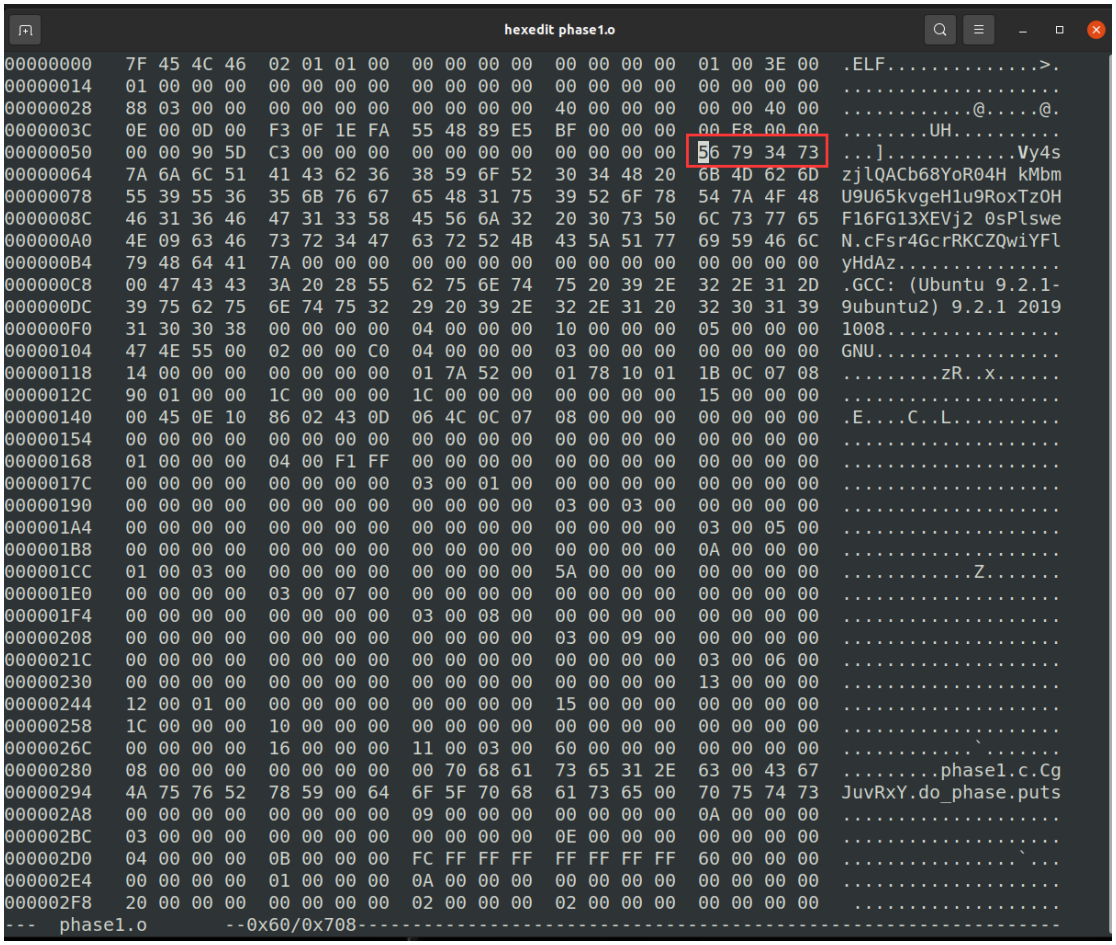
然后再用 readelf -r 看一眼 phase1.o 的重定位信息：

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 readelf -r phase1.o

重定位节 '.rela.text' at offset 0x2b0 contains 2 entries:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000000009  00030000000a R_X86_64_32       0000000000000000 .data + 0
00000000000e  000b00000004 R_X86_64_PLT32    0000000000000000 puts - 4

重定位节 '.rela.data' at offset 0x2e0 contains 1 entry:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000000060  000a00000001 R_X86_64_64       0000000000000000 do_phase + 0

重定位节 '.rela.eh_frame' at offset 0x2f8 contains 1 entry:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
```

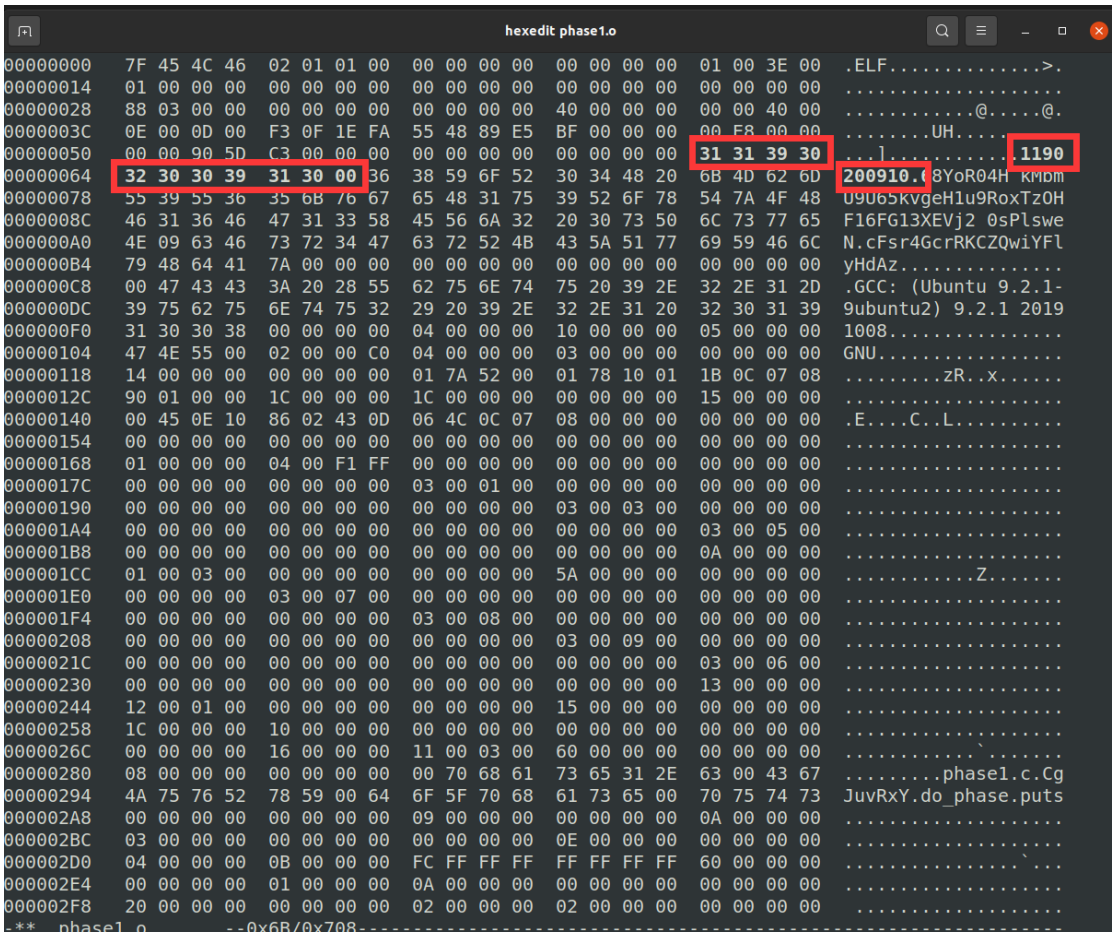
用红框框起来的这一条重定位条目，应该就是 puts 输出的字符串的所在地址，它位于.data 节偏移量 0 的位置，也就是我们需要从.data 节偏移量 0 的地方修改。

综合上面的分析，.data 节从 phase1.o 文件的第 0x60 个字节开始，我们需要修改.data 节偏移量为 0 的字符串。

然后我们开始使用 hexedit：



框起来的部分是 phase1.o 文件的第 0x60 个字节，也就是.data 节开始的地方，我们需要修改这里保存的字符串，将它改为“1190200910”，用 ASCII 码写入。注意字符串要以 0 结尾。



写入完成后按下 Ctrl+W 保存，按 Ctrl+X 退出 hexedit。

然后再重新链接并运行程序，可以看到已经输出我的学号。

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 ls
main.o  phase1.o  phase1.out  phase1.s  phase2.o  phase3.o  phase4.o  phase5.o
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 gcc -no-pie -o phase1_result m
ain.o phase1.o
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 ./phase1_result
1190200910
```

3.2 阶段 2 的分析

程序运行结果截图：

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 gcc -no-pie -o phase2_result
main.o phase2.o
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 ./phase2_result
1190200910
```

分析与设计的过程：

先通过 objdump -d 看一下 phase2.o 的反汇编：

```
yx1190200910@icsUbuntu2004 E ~/Desktop/ICS/lab5/linklab-1190200910 E objdump -d phase2.o

phase2.o:     文件格式 elf64-x86-64


Disassembly of section .text:

0000000000000000 <zKlIjzNp>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 83 ec 10             sub    $0x10,%rsp
   c:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
  10:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  14:   be 00 00 00 00          mov    $0x0,%esi
  19:   48 89 c7                mov    %rax,%rdi
  1c:   e8 00 00 00 00          callq  21 <zKlIjzNp+0x21>
  21:   85 c0                   test   %eax,%eax
  23:   75 0e                   jne    33 <zKlIjzNp+0x33>
  25:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  29:   48 89 c7                mov    %rax,%rdi
  2c:   e8 00 00 00 00          callq  31 <zKlIjzNp+0x31>
  31:   eb 01                   jmp    34 <zKlIjzNp+0x34>
  33:   90                      nop
  34:   c9                      leaveq
  35:   c3                      retq

0000000000000036 <do_phase>:
  36:   f3 0f 1e fa             endbr64
  3a:   55                      push   %rbp
  3b:   48 89 e5                mov    %rsp,%rbp
  3e:   90                      nop
  3f:   90                      nop
  40:   90                      nop
  41:   90                      nop
  42:   90                      nop
```

我们需要在 do_phase 内部调用上面的 zKlIjzNp 函数从而完成学号的输出。因此我们需要编辑.text 段的 do_phase 代码。

用 readelf -S 看一下 phase2.o 的节头信息：

```
yx1190200910@icsUbuntu2004 E ~/Desktop/ICS/lab5/linklab-1190200910 E readelf -S phase2.o
There are 15 section headers, starting at offset 0x400:

节头：
  [号] 名称              类型             地址              偏移量
       大小              全体大小          旗标    链接    信息    对齐
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       0000000000000061  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000210
       0000000000000048  0000000000000018   I      12     1     8
  [ 3] .data             PROGBITS         0000000000000000  000000a8
       0000000000000008  0000000000000000  WA       0     0     8
  [ 4] .rela.data        RELA             0000000000000000  00000338
       0000000000000018  0000000000000018   I      12     3     8
```

可以看到.text 节的文件偏移是 0x40 字节，也就是说.text 节从文件的第 0x40 个字节开始。

然后用 readelf -r 看一下 phase2.o 的重定位信息：

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 readelf -r phase2.o

重定位节 '.rela.text' at offset 0x2f0 contains 3 entries:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000000015  00050000000a R_X86_64_32       0000000000000000 .rodata + 0
00000000001d  000b00000004 R_X86_64_PLT32    0000000000000000 strcmp - 4
00000000002d  000c00000004 R_X86_64_PLT32    0000000000000000 puts - 4

重定位节 '.rela.data' at offset 0x338 contains 1 entry:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000000000  000d00000001 R_X86_64_64       0000000000000036 do_phase + 0

重定位节 '.rela.eh_frame' at offset 0x350 contains 2 entries:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000000020  000200000002 R_X86_64_PC32     0000000000000000 .text + 0
000000000040  000200000002 R_X86_64_PC32     0000000000000000 .text + 36
```

下方的.text+0 和.text+36 应该就是 phase2.o 中的两个函数，其中第一个函数的入口在.text 节的偏移 0 的位置，第二个函数的入口在.text 节偏移 0x36 的位置，结合刚刚 objdump -d 的结果，可以判断出 do_phase 函数的入口就是位于.text 节偏移 0x36 位置。

但我们还不知道如何编写在 do_phase 函数中填入的机器代码，我们先把原始的 phase2.o 和 main.o 链接，然后用 EDB 调试，在 EDB 中编写汇编代码并编译成机器语言，我们复制对应的机器码即可。

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 gcc -no-pie -o linked_phase2 main.o phase2.o
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 ls
linked_phase2  main.o  phase1.o  phase1_result  phase2.o  phase3.o  phase4.o  phase5.o
```

链接 phase2.o，然后用 EDB 运行上一步的结果 linked_phase2。

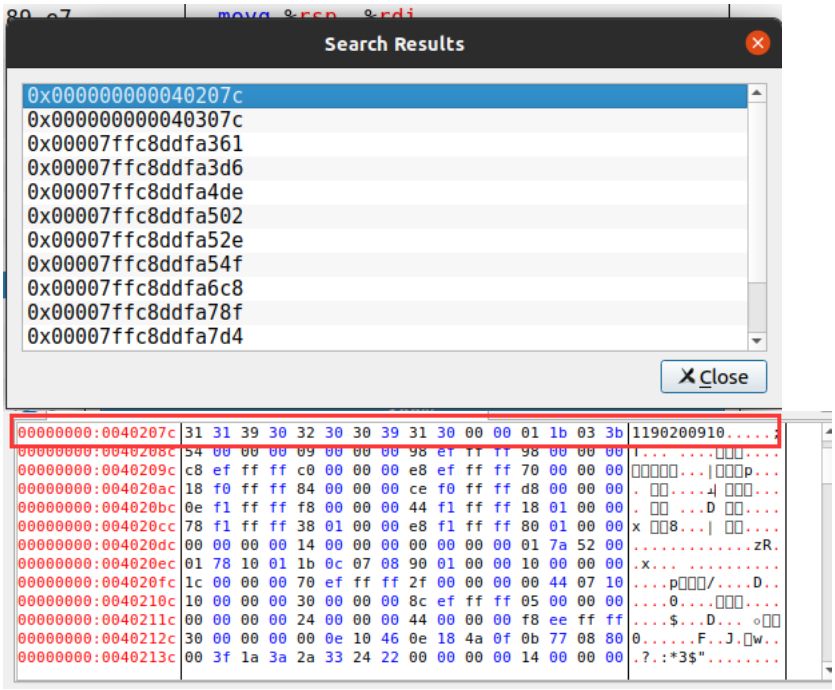
```
00000000:00401196 linked_phase2!zKlIjzNp   f3              db 0xf3
00000000:00401197                          0f              db 0x0f
00000000:00401198                          1e              db 0x1e
00000000:00401199                          fa              cli
00000000:0040119a                          55              pushq %rbp
00000000:0040119b                          48 89 e5        movq %rsp, %rbp
00000000:0040119e                          48 83 ec 10     subq $0x10, %rsp
00000000:004011a2                          48 89 7d f8     movq %rdi, -8(%rbp)
00000000:004011a6                          48 8b 45 f8     movq -8(%rbp), %rax
00000000:004011aa                          be 7c 20 40 00  movl $0x40207c, %esi
00000000:004011af                          48 89 c7        movq %rax, %rdi
00000000:004011b2                          e8 a9 fe ff ff  callq 0x401060
00000000:004011b7                          85 c0           testl %eax, %eax
00000000:004011b9                          75 0e           jne 0x4011c9
00000000:004011bb                          48 8b 45 f8     movq -8(%rbp), %rax
00000000:004011bf                          48 89 c7        movq %rax, %rdi
00000000:004011c2                          e8 89 fe ff ff  callq linked_phase2!.plt+0x30
00000000:004011c7                          eb 01           jmp 0x4011ca
00000000:004011c9                          90              nop
00000000:004011ca                          c9              leave
00000000:004011cb                          c3              retq
00000000:004011cc linked_phase2!do_phase   f3              db 0xf3
00000000:004011cd                          0f              db 0x0f
00000000:004011ce                          1e              db 0x1e
00000000:004011cf                          fa              cli
00000000:004011d0                          55              pushq %rbp
00000000:004011d1                          48 89 e5        movq %rsp, %rbp
00000000:004011d4                          90              nop
00000000:004011d5                          90              nop
00000000:004011d6                          90              nop
```

可以看到，do_phase 中有一系列的 nop 指令，我们需要在其中 call zKlIjzNp 函数。利用 EDB 的 assemble 功能编写汇编代码并编译，得到这样的结果：

```
00000000:004011cc linked_phase2!do_phase   f3              db 0xf3
00000000:004011cd                          0f              db 0x0f
00000000:004011ce                          1e              db 0x1e
00000000:004011cf                          fa              cli
00000000:004011d0                          55              pushq %rbp
00000000:004011d1                          48 89 e5        movq %rsp, %rbp
00000000:004011d4                          e8 bd ff ff ff  callq linked_phase2!zKlIjzNp
00000000:004011d9                          90              nop
```

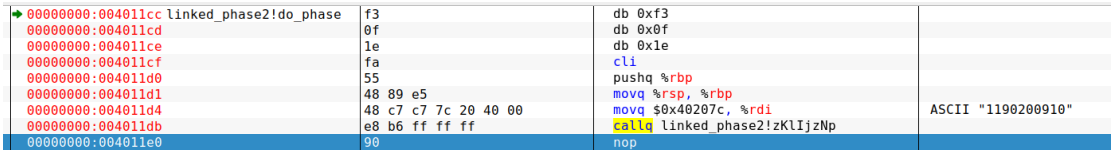
但是，zKlIjzNp 需要接收一个参数，该参数是输出字符串的所在地址，因此在程序中需要找到自己学号字符串的地址。我们在 EDB 中直接 Ctrl+F 查找 1190200910，

可以查到以下结果。



说明程序中 0x40207c 的地址处就存放着我学号的字符串。为了验证这个地址是否会动态变化，多次运行 EDB，发现这个地址不会改变，每次都是 0x40207c。

因此重新编写 do_phase 的汇编代码：



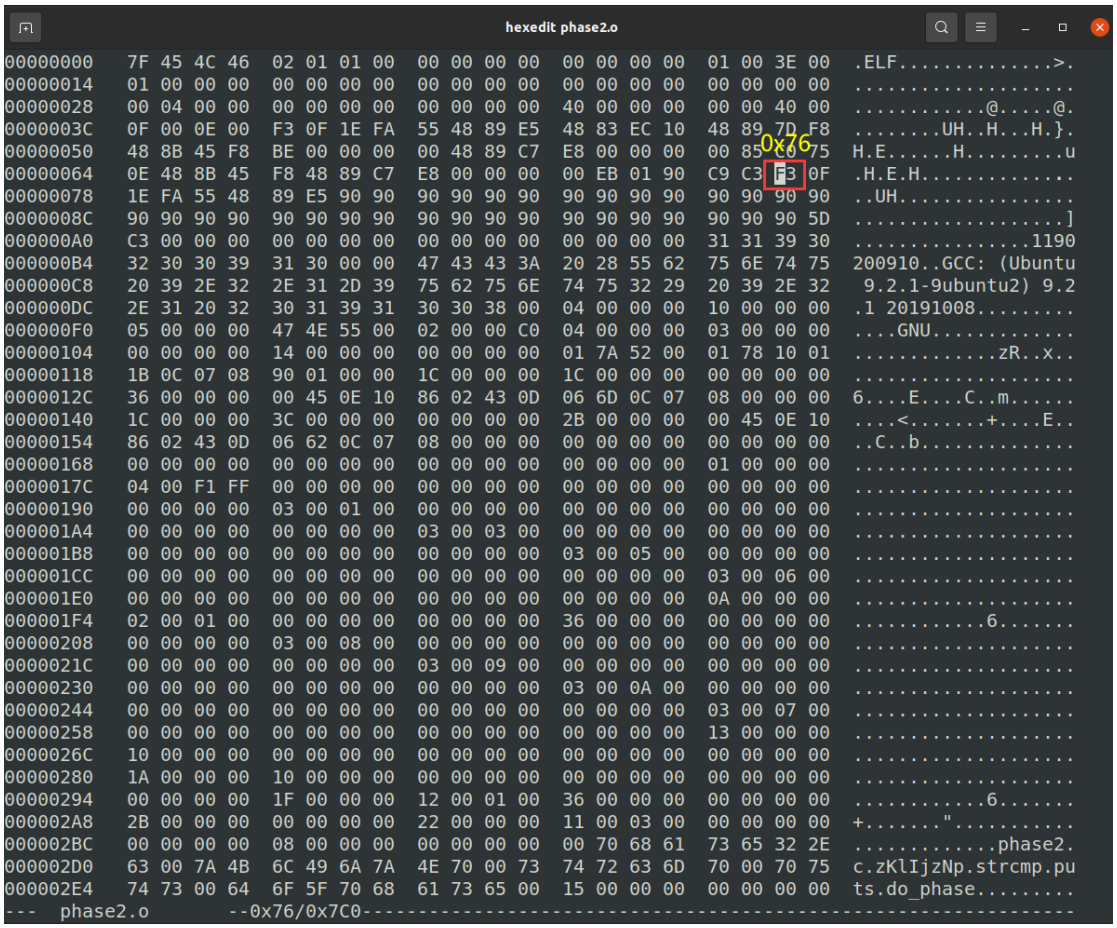
经过测试这段代码可以让 do_phase 调用 zKlIjzNp 函数并输出学号。

因此我们需要在 phase2.o 这个 ELF 文件中找到.text 段(文件偏移为 0x40)，在.text 段中找到 do_phase 函数入口(段内偏移为 0x36)，然后将 do_phase 函数的 nop 指令(机器码 90)改为下面的机器码：

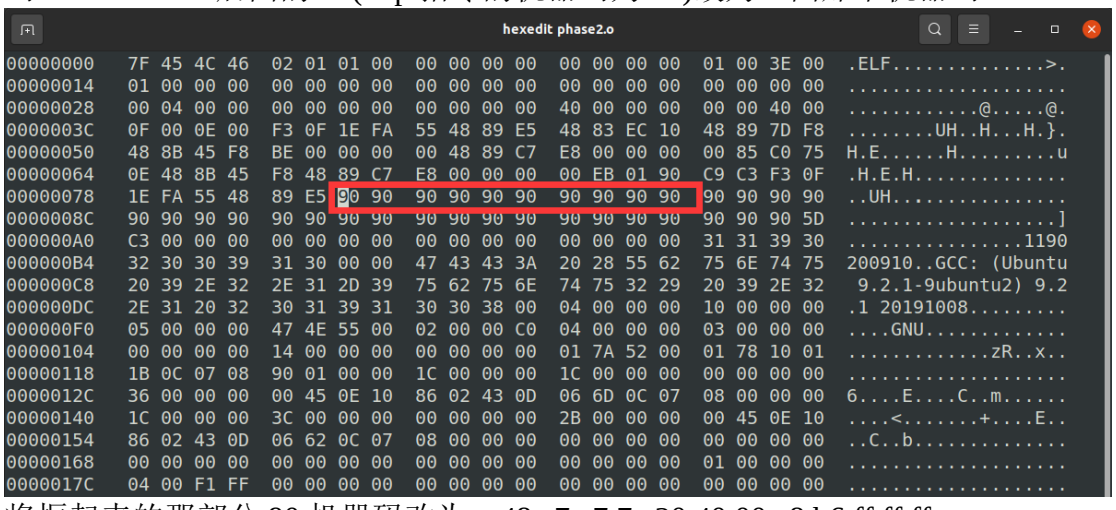
48 c7 c7 7c 20 40 00

e8 b6 ff ff ff

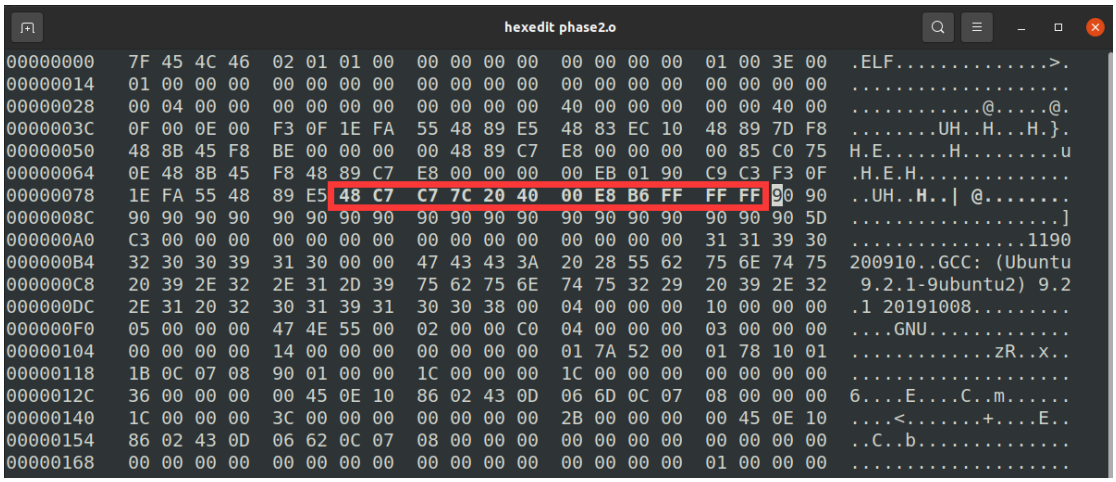
利用 hexedit 编辑 phase2.o。经过计算，phase2.o 中 do_phase 函数入口与文件开头偏移量为 0x40+0x36=0x76。



图中框起的部分即为文件偏移 0x76 处，也就是 do_phase 函数的入口，这部分机器码“F3 0F 1E FA……”与在 EDB 调试中看到的机器码是一致的。我们需要将机器码“48 89 E5”后面的 90(nop 指令的机器码为 90)改为上面那串机器码。

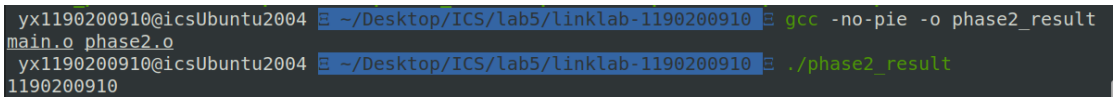


将框起来的那部分 90 机器码改为：48 c7 c7 7c 20 40 00 e8 b6 ff ff ff



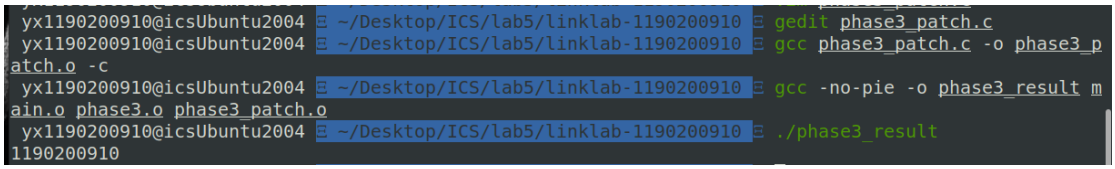
修改后的结果如上图所示。按 Ctrl+W 保存，按 Ctrl+X 退出。

然后再次编译运行，可以看到程序正确输出了学号。



3.3 阶段 3 的分析

程序运行结果截图：



分析与设计的过程：

```
char PHASE3_CODEBOOK[256];
void do_phase(){
    const char char cookie[] = PHASE3_COOKIE;
    for( int i=0; i<sizeof(cookie)-1; i++ )
        printf( "%c", PHASE3_CODEBOOK[ (unsigned char)(cookie[i]) ] );
    printf( "\n" );
}
```

分析 phase3 的源码可知，其中定义了一个未初始化的全局数组 PHASE3_CODEBOOK，这显然是一个弱符号，因此我们可以在 phase3_patch.o 中定义一个同名的符号并初始化，初始化之后这就是一个强符号，在链接时让链接器选择我们自己构造的强符号。因此剩下的问题就是，如何巧妙的构造这个数组，

使其输出自己的学号。
我的学号是 1190200910，因此要求构造的数组满足：
PHASE3_CODEBOOK [(unsigned char)(cookie[0])] = '1' = 31
PHASE3_CODEBOOK [(unsigned char)(cookie[1])] = '1' = 31
PHASE3_CODEBOOK [(unsigned char)(cookie[2])] = '9' = 39
PHASE3_CODEBOOK [(unsigned char)(cookie[3])] = '0' = 30
PHASE3_CODEBOOK [(unsigned char)(cookie[4])] = '2' = 32
PHASE3_CODEBOOK [(unsigned char)(cookie[5])] = '0' = 30
PHASE3_CODEBOOK [(unsigned char)(cookie[6])] = '0' = 30
PHASE3_CODEBOOK [(unsigned char)(cookie[7])] = '9' = 39
PHASE3_CODEBOOK [(unsigned char)(cookie[8])] = '1' = 31
PHASE3_CODEBOOK [(unsigned char)(cookie[9])] = '0' = 30

为了构造 PHASE3_CODEBOOK 数组，我们必须知道 cookie 指向的那个数组中的内容。因此我们先用原始的 phase3.o 和 main.o 链接，并用 EDB 调试这个程序。

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 gcc -no-pie -o linked_phase3 m
ain.o phase3.o
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 ls
linked_phase3  phase1.o       phase2.o       phase3.o  phase5.o
main.o         phase1_result  phase2_result  phase4.o
```

```
xorl %eax, %eax
movabsq $0x7470667771766e61, %rax
movq %rax, -0x13(%rbp)
movw $0x7365, -0xb(%rbp)
movb $0, -9(%rbp)
movl $0, -0x18(%rbp)
jmp 0x401216
```

在 do_phase 的汇编中看到两处奇怪的立即数，两个立即数放在了从 rbp-0x13 开始的连续 10 个字节内。而我们的学号就是 10 个字节的，因此怀疑这就是 do_phase 函数一开始给 cookie 数组所赋的值。10 个字节数分别是：0x74, 0x70, 0x66, 0x77, 0x71, 0x76, 0x6e, 0x61, 0x73, 0x65。
通过 do_phase 下面的汇编代码分析，这一处就是 for 循环的代码。

```
00000000:004011f0  eb 24           jmp 0x401216
00000000:004011f2  8b 45 e8        movl -0x18(%rbp), %eax
00000000:004011f5  48 98           cltq
00000000:004011f7  0f b6 44 05 ed  movzbl -0x13(%rbp, %rax), %eax
00000000:004011fc  0f b6 c0        movzbl %al, %eax
00000000:004011ff  48 98           cltq
00000000:00401201  0f b6 80 80 40 40 00  movzbl 0x404080(%rax), %eax
00000000:00401208  0f be c0        movsbl %al, %eax
00000000:0040120b  89 c7           movl %eax, %edi
00000000:0040120d  e8 4e fe ff ff  callq linked_phase3!.plt+0x40
00000000:00401212  83 45 e8 01     addl $1, -0x18(%rbp)
00000000:00401216  8b 45 e8        movl -0x18(%rbp), %eax
00000000:00401219  83 f8 09        cmpl $9, %eax
00000000:0040121c  76 d4           jbe 0x4011f2
```

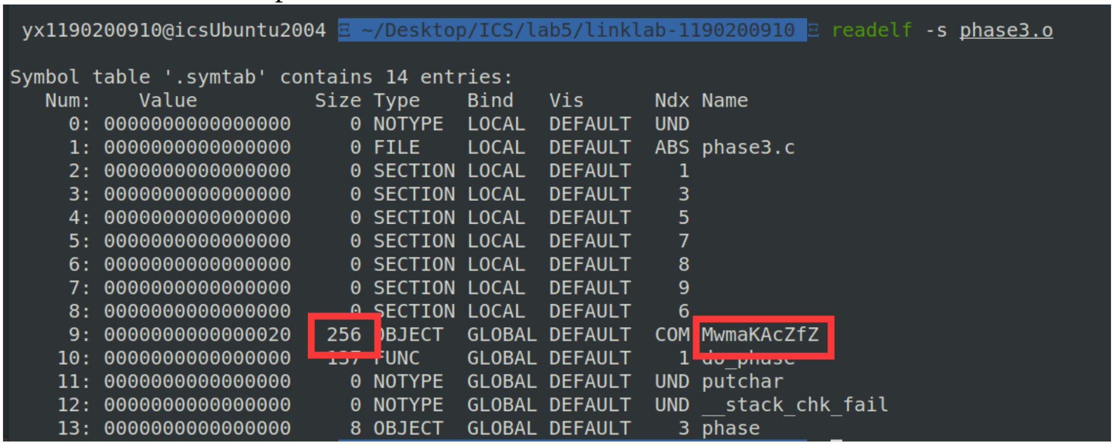
在 rbp-0x18 处保存了循环变量 i，如果 i<=9 就会以 i 为变址，从 rbp-0x13 处进行寻址，并把数组寻址的结果放到 eax 中，这也就印证了我们上面的推测。因此现在可以确定，cookie 数组中的元素为：{0x74, 0x70, 0x66, 0x77, 0x71, 0x76, 0x6e, 0x61, 0x73, 0x65}。
因此我们需要构造的 PHASE3_CODEBOOK 数组必须满足：
PHASE3_CODEBOOK[0x74]='1'=0x31
PHASE3_CODEBOOK[0x70]='1'=0x31
PHASE3_CODEBOOK[0x66]='9'=0x39
PHASE3_CODEBOOK[0x77]='0'=0x30

PHASE3_CODEBOOK[0x71]='2'=0x32
PHASE3_CODEBOOK[0x76]='0'=0x30
PHASE3_CODEBOOK[0x6e]='0'=0x30
PHASE3_CODEBOOK[0x61]='9'=0x39
PHASE3_CODEBOOK[0x73]='1'=0x31
PHASE3_CODEBOOK[0x65]='0'=0x30

于是，定义数组：

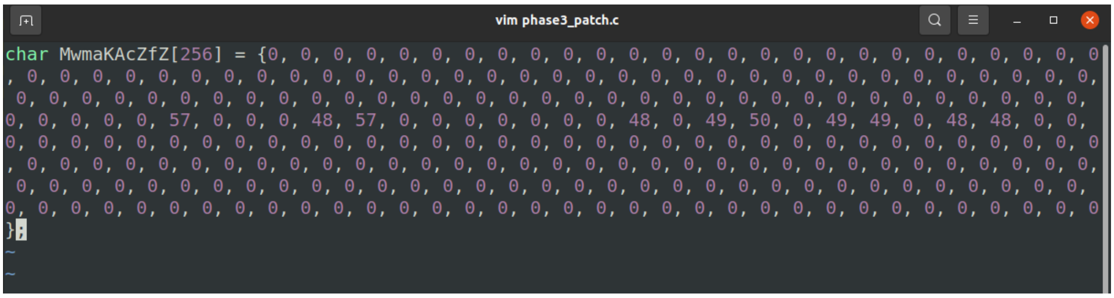
char PHASE3_CODEBOOK[256] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 57, 0, 0, 0, 48, 57, 0, 0, 0, 0, 0, 0, 0, 48, 0, 49, 50, 0, 49, 49, 0, 48, 48, 0};

利用 readelf -s 查看 phase3.o 的符号表：



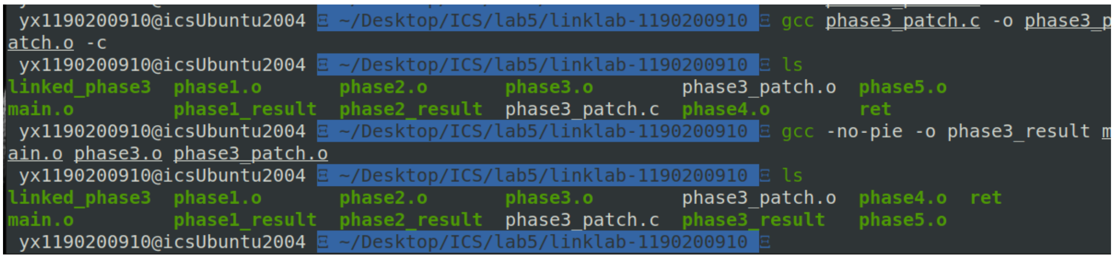
```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 readelf -s phase3.o

Symbol table '.symtab' contains 14 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 0000000000000000     0 FILE    LOCAL  DEFAULT  ABS phase3.c
     2: 0000000000000000     0 SECTION LOCAL  DEFAULT    1
     3: 0000000000000000     0 SECTION LOCAL  DEFAULT    3
     4: 0000000000000000     0 SECTION LOCAL  DEFAULT    5
     5: 0000000000000000     0 SECTION LOCAL  DEFAULT    7
     6: 0000000000000000     0 SECTION LOCAL  DEFAULT    8
     7: 0000000000000000     0 SECTION LOCAL  DEFAULT    9
     8: 0000000000000000     0 SECTION LOCAL  DEFAULT    6
     9: 0000000000000020   256 OBJECT  GLOBAL DEFAULT  COM MwmaKAcZfZ
    10: 0000000000000000   157 FUNC    GLOBAL DEFAULT    1 do_phase
    11: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND putchar
    12: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT  UND __stack_chk_fail
    13: 0000000000000000     8 OBJECT  GLOBAL DEFAULT    3 phase
```

其中 MwmaKAcZfZ 即为 PHASE3_CODEBOOK 的名字。因此我们在 phase3_patch.c 文件中也定义一个同名的数组，并且把上面计算出的数组元素值给赋进去：



```
vim phase3_patch.c
char MwmaKAcZfZ[256] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 57, 0, 0, 0, 48, 57, 0, 0, 0, 0, 0, 0, 0, 48, 0, 49, 50, 0, 49, 49, 0, 48, 48, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
};
~
~
```

然后先编译 phase3_patch（不链接），然后将 main.o，phase3.o，phase3_patch.o 三个文件一起链接。



最后运行生成的程序：

发现结果不是正确的学号。顺序发生了错误，前 8 个数字倒序了，后 2 个数字也倒序了。

回过头来找错误原因，发现是分析 do_phase 代码时，给 cookie 数组赋初值时，应该是小端序存储数据，而我跟随思维定势当作大端序，从低地址到高地址赋值了。其中第一个立即数是 0x7470667771766e61，转换成小端字节，从低地址到高地址就是：0x61, 0x6e, 0x76, 0x71, 0x77, 0x66, 0x70, 0x74；第二个立即数是 0x7365，转化为小端字节，从低地址到高地址为 0x65, 0x73。

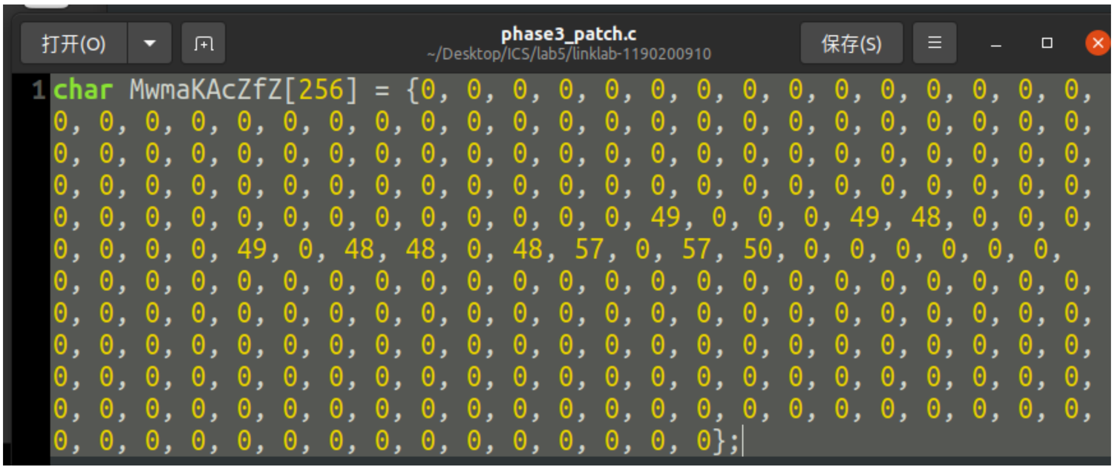
于是修改 PHASE3_CODEBOOK 数组中的赋值顺序：

PHASE3_CODEBOOK[0x61]=0x31;
PHASE3_CODEBOOK[0x6e]= 0x31;
PHASE3_CODEBOOK[0x76]=0x39;
PHASE3_CODEBOOK[0x71]=0x30;
PHASE3_CODEBOOK[0x77]=0x32;
PHASE3_CODEBOOK[0x66]=0x30;
PHASE3_CODEBOOK[0x70]=0x30;
PHASE3_CODEBOOK[0x74]=0x39;
PHASE3_CODEBOOK[0x65]=0x31;
PHASE3_CODEBOOK[0x73]=0x30;

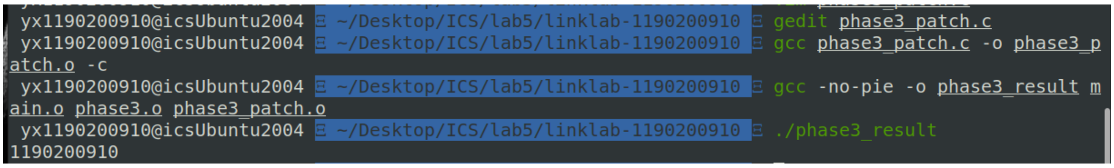
于是得到新的数组：

char PHASE3_CODEBOOK[256] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 49, 0, 0, 0, 49, 48, 0, 0, 0, 0, 0, 0, 49, 0, 48, 48, 0, 48, 57, 0, 57, 50, 0};

据此对 phase3_patch.c 进行修改：



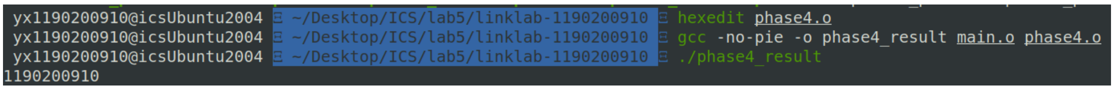
修改后重新编译、链接、运行：



经过上面多步的分析、调试，修改，终于能输出正确的学号！

3.4 阶段 4 的分析

程序运行结果截图：



分析与设计的过程：

phase4.c程序框架

```
void do_phase()
{
    const char cookie[] = PHASE4_COOKIE;
    char c;
    for ( int i = 0; i < sizeof(cookie)-1; i++ )
    {
        c = cookie[i];
        switch (c)
        {
            // 每个学生的映射关系和case顺序建议不一样
            case ‘A’: { c = 48; break; }
            case ‘B’: { c = 121; break; }
             …
            case ‘Z’: { c = 93; break; }
        }
        printf("%c", c);
    }
}
```

这时 phase4 的源码。这一关要求我们修改 switch 跳转表使得程序输出自己的学号。我们首先需要知道 cookie 数组中保存的元素值，就先用原始的 phase4.o 和 main.o 链接得到可执行文件，然后用 EDB 调试它，观察 cookie 数组的元素。

```
linked_phase4!do_phase  f3                      db 0xf3
                        0f                      db 0x0f
                        1e                      db 0x1e
                        fa                      cli
                        55                      pushq %rbp
                        48 89 e5                movq %rsp, %rbp
                        48 83 ec 20             subq $0x20, %rsp
                        64 48 8b 04 25 28 00 00 00  movq %fs:0x28, %rax
                        48 89 45 f8             movq %rax, -8(%rbp)
                        31 c0                   xorl %eax, %eax
                        48 b8 43 4a 4d 57 46 44 56 …  movabsq $0x52564446574d4a43, %rax
                        48 89 45 ed             movq %rax, -0x13(%rbp)
                        66 c7 45 f5 4e 54       movw $0x544e, -0xb(%rbp)
                        c6 45 f7 00             movb $0, -9(%rbp)
                        c7 45 e8 00 00 00 00    movl $0, -0x18(%rbp)
                        e9 e0 00 00 00          jmp 0x4012d5
                        8b 45 e8                movl -0x18(%rbp), %eax
                        48 98                   cltq
                        0f b6 44 05 ed          movzbl -0x13(%rbp, %rax), %eax
```

与上一关相同，汇编代码仍然采用立即数的方式给 cookie 数组赋值，由于是采用小端序存储，cookie 数组中的元素分别为：{0x43, 0x4a, 0x4d, 0x57, 0x46, 0x44, 0x56, 0x52, 0x4e, 0x54}，用 ASCII 码转成字符就是：{‘C’, ‘J’, ‘M’, ‘W’, ‘F’, ‘D’, ‘V’, ‘R’, ‘N’, ‘T’}

直接运行一下这个程序，可以看到最后的输出是：

Nr}C012;`6

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 ./linked_phase4
Nr}C012;`6
```

由此可以推断出之前的跳转表：

```
switch (c){
   case 'C':
{
      c = 'N';
      break;
   }
   case 'J':
   {
      c = 'r';
      break;
   }
   case 'M':
   {
      c = '}';
      break;
   }
   case 'W':
   {
      c = 'C';
      break;
   }
   case 'F':
```

```
   {
      c = '0';
      break;
   }
   case 'D':
   {
      c = '1';
      break;
   }
   case 'V':
   {
      c = '2';
      break;
   }
   case 'R':
   {
      c = ';';
      break;
   }
   case 'N':
   {
      c = '`';
      break;
   }
   case 'T':
   {
      c = '6';
      break;
   }
   }
```

序列化并转成 ASCII 码表示：

```
switch (c)
   {
   case 'C':
   {
      c = 0x4e;
      break;
   }
   case 'D':
   {
      c = 0x31;
      break;
   }
   case 'F':
   {
      c = 0x30;
      break;
```

```
}

case 'J':
{
   c = 0x72;
   break;
}
case 'M':
{
   c = 0x7d;
   break;
}
case 'N':
{
   c = 0x60;
   break;
}
case 'R':
{
   c = 0x3d;
   break;
}
case 'T':
{
   c = 0x36;
   break;
}
case 'V':
{
   c = 0x32;
   break;
}
case 'W':
{
   c = 0x43;
   break;
}
}
```

然后通过 readelf 读取 phase4.o，在 rodata 节寻找有上面 switch 跳转表特征的部分，从而确定 switch 跳转表在.rodata 节的偏移。

通过 readelf -S 看一下 phase4.o 的节头信息：

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 readelf -S phase4.o
There are 16 section headers, starting at offset 0x7f0:

节头：
  [号] 名称              类型             地址              偏移量
       大小              全体大小         旗标    链接    信息    对齐
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       000000000000014c  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000470
       0000000000000060  0000000000000018   I      13     1     8
  [ 3] .data             PROGBITS         0000000000000000  00000190
       0000000000000008  0000000000000000  WA       0     0     8
  [ 4] .rela.data        RELA             0000000000000000  000004d0
       0000000000000018  0000000000000018   I      13     3     8
  [ 5] .bss              NOBITS           0000000000000000  00000198
       0000000000000000  0000000000000000  WA       0     0     1
  [ 6] .rodata           PROGBITS         0000000000000000  00000198
       00000000000000d0  0000000000000000  A        0     0     8
  [ 7] .rela.rodata      RELA             0000000000000000  000004e8
       0000000000000270  0000000000000018   I      13     6     8
  [ 8] .comment          PROGBITS         0000000000000000  00000268
       000000000000002d  0000000000000001  MS       0     0     1
```

可以发现.rela.rodata 节（只读节的重定位节）开始于文件偏移 0x4e8 的位置。

通过 readelf -r 读取 phase4.o 的重定位信息，可以看到在.rodata 节有 26 个重定位条目：

```
重定位节 '.rela.rodata' at offset 0x4e8 contains 26 entries:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000000000  000200000001 R_X86_64_64       0000000000000000 .text + 69
000000000008  000200000001 R_X86_64_64       0000000000000000 .text + 72
000000000010  000200000001 R_X86_64_64       0000000000000000 .text + 7b
000000000018  000200000001 R_X86_64_64       0000000000000000 .text + 84
000000000020  000200000001 R_X86_64_64       0000000000000000 .text + 8d
000000000028  000200000001 R_X86_64_64       0000000000000000 .text + 93
000000000030  000200000001 R_X86_64_64       0000000000000000 .text + 99
000000000038  000200000001 R_X86_64_64       0000000000000000 .text + 9f
000000000040  000200000001 R_X86_64_64       0000000000000000 .text + a5
000000000048  000200000001 R_X86_64_64       0000000000000000 .text + ab
000000000050  000200000001 R_X86_64_64       0000000000000000 .text + b1
000000000058  000200000001 R_X86_64_64       0000000000000000 .text + b7
000000000060  000200000001 R_X86_64_64       0000000000000000 .text + bd
000000000068  000200000001 R_X86_64_64       0000000000000000 .text + c3
000000000070  000200000001 R_X86_64_64       0000000000000000 .text + c9
000000000078  000200000001 R_X86_64_64       0000000000000000 .text + cf
000000000080  000200000001 R_X86_64_64       0000000000000000 .text + d5
000000000088  000200000001 R_X86_64_64       0000000000000000 .text + db
000000000090  000200000001 R_X86_64_64       0000000000000000 .text + e1
000000000098  000200000001 R_X86_64_64       0000000000000000 .text + e7
0000000000a0  000200000001 R_X86_64_64       0000000000000000 .text + ed
0000000000a8  000200000001 R_X86_64_64       0000000000000000 .text + f3
0000000000b0  000200000001 R_X86_64_64       0000000000000000 .text + f9
0000000000b8  000200000001 R_X86_64_64       0000000000000000 .text + ff
0000000000c0  000200000001 R_X86_64_64       0000000000000000 .text + 105
0000000000c8  000200000001 R_X86_64_64       0000000000000000 .text + 10b
```

这正好与 switch 跳转表的 26 个字母相对应，因此推断，这 26 个重定位条目就是 switch 跳转表，它开始于.rodata 节偏移 0 的位置。

这些跳转应该就是从 case ‘A’到 case ‘Z’每一个跳转，跳转到的.text 节位置。

根据 cookie 数组的内容{‘C’, ‘J’, ‘M’, ‘W’, ‘F’, ‘D’, ‘V’, ‘R’, ‘N’, ‘T’}，我们需要让

跳转表满足以下要求：
case ‘C’跳转到 c = 0x31
case ‘J’跳转到 c = 0x31
case ‘M’跳转到 c = 0x39
case ‘W’跳转到 c = 0x30
case ‘F’跳转到 c = 0x32
case ‘D’跳转到 c = 0x30
case ‘V’跳转到 c = 0x30
case ‘R’跳转到 c = 0x39
case ‘N’跳转到 c = 0x31
case ‘T’跳转到 c = 0x30
通过 objdump -d 查看 phase4.o 中 text 节的反汇编指令：

```
0000000000000000 <do_phase>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   48 83 ec 20             sub    $0x20,%rsp
   c:   64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  13:   00 00
  15:   48 89 45 f8             mov    %rax,-0x8(%rbp)
  19:   31 c0                   xor    %eax,%eax
  1b:   48 b8 43 4a 4d 57 46    movabs $0x52564446574d4a43,%rax
  22:   44 56 52
  25:   48 89 45 ed             mov    %rax,-0x13(%rbp)
  29:   66 c7 45 f5 4e 54       movw   $0x544e,-0xb(%rbp)
  2f:   c6 45 f7 00             movb   $0x0,-0x9(%rbp)
  33:   c7 45 e8 00 00 00 00    movl   $0x0,-0x18(%rbp)
  3a:   e9 e0 00 00 00          jmpq   11f <do_phase+0x11f>
  3f:   8b 45 e8                mov    -0x18(%rbp),%eax
  42:   48 98                   cltq
  44:   0f b6 44 05 ed          movzbl -0x13(%rbp,%rax,1),%eax
  49:   88 45 e7                mov    %al,-0x19(%rbp)
  4c:   0f be 45 e7             movsbl -0x19(%rbp),%eax
  50:   83 e8 41                sub    $0x41,%eax
  53:   83 f8 19                cmp    $0x19,%eax
  56:   0f 87 b4 00 00 00       ja     110 <do_phase+0x110>
  5c:   89 c0                   mov    %eax,%eax
  5e:   48 8b 04 c5 00 00 00    mov    0x0(,%rax,8),%rax
  65:   00
  66:   3e ff e0                notrack jmpq *%rax
  69:   c6 45 e7 39             movb   $0x39,-0x19(%rbp)
  6d:   e9 9e 00 00 00          jmpq   110 <do_phase+0x110>
  72:   c6 45 e7 3f             movb   $0x3f,-0x19(%rbp)
  76:   e9 95 00 00 00          jmpq   110 <do_phase+0x110>
  7b:   c6 45 e7 4e             movb   $0x4e,-0x19(%rbp)
  7f:   e9 8c 00 00 00          jmpq   110 <do_phase+0x110>
  84:   c6 45 e7 31             movb   $0x31,-0x19(%rbp)
  88:   e9 83 00 00 00          jmpq   110 <do_phase+0x110>
  8d:   c6 45 e7 63             movb   $0x63,-0x19(%rbp)
  91:   eb 7d                   jmp    110 <do_phase+0x110>
  93:   c6 45 e7 30             movb   $0x30,-0x19(%rbp)
  97:   eb 77                   jmp    110 <do_phase+0x110>
  99:   c6 45 e7 33             movb   $0x33,-0x19(%rbp)
```

由于 do_phase 函数入口位于.text 节偏移量为 0 的位置，因此跳转表中的相对.text 节的偏移量就相当于反汇编指令中相对于 do_phase 函数入口的偏移量。

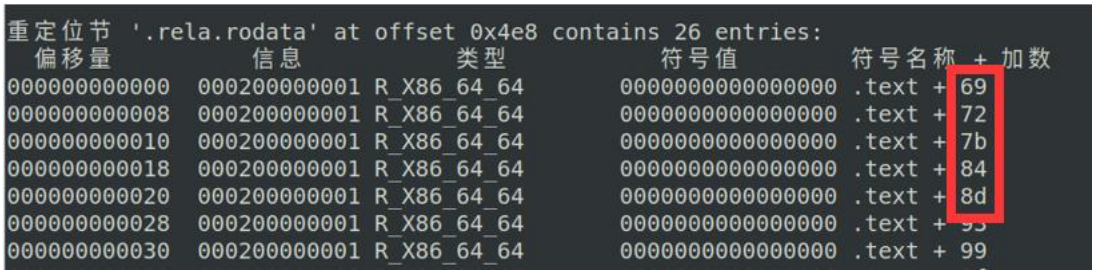
```
 69:   c6 45 e7 39             movb   $0x39,-0x19(%rbp)
 72:   c6 45 e7 3f             movb   $0x3f,-0x19(%rbp)
 76:   e9 95 00 00 00          jmpq   110 <do_phase+0x110>
 7b:   c6 45 e7 4e             movb   $0x4e,-0x19(%rbp)
 84:   c6 45 e7 31             movb   $0x31,-0x19(%rbp)
 88:   e9 83 00 00 00          jmpq   110 <do_phase+0x110>
 8d:   c6 45 e7 63             movb   $0x63,-0x19(%rbp)
 93:   c6 45 e7 30             movb   $0x30,-0x19(%rbp)
 97:   eb 77                   jmp    110 <do_phase+0x110>
 f3:   c6 45 e7 32             movb   $0x32,-0x19(%rbp)
 f9:   c6 45 e7 43             movb   $0x43,-0x19(%rbp)
 fd:   eb 11                   jmp    110 <do_phase+0x110>
 ff:   c6 45 e7 6f             movb   $0x6f,-0x19(%rbp)
103:   eb 0b                   jmp    110 <do_phase+0x110>
105:   c6 45 e7 74             movb   $0x74,-0x19(%rbp)
```

通过分析反汇编代码，可以看到，
将 c 赋值为 0x31 位于偏移量为 0x84 的位置
将 c 赋值为 0x39 位于偏移量为 0x69 的位置
将 c 赋值为 0x30 位于偏移量为 0x93 的位置
将 c 赋值为 0x32 位于偏移量为 0xf3 的位置

通过 readelf -x.rela.rodata phase4.o 查看 phase4.o 的.rodata 节内容：

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 readelf -x.rela.rodata phase4.o

".rela.rodata"节的十六进制输出:
  0x00000000 00000000 00000000 01000000 02000000 ................
  0x00000010 69000000 00000000 08000000 00000000 i...............
  0x00000020 01000000 02000000 72000000 00000000 ........r.......
  0x00000030 10000000 00000000 01000000 02000000 ................
  0x00000040 7b000000 00000000 18000000 00000000 {...............
  0x00000050 01000000 02000000 84000000 00000000 ................
  0x00000060 20000000 00000000 01000000 02000000  ...............
  0x00000070 8d000000 00000000 28000000 00000000 ........(.......
  0x00000080 01000000 02000000 93000000 00000000 ................
  0x00000090 30000000 00000000 01000000 02000000 0...............
  0x000000a0 99000000 00000000 38000000 00000000 ........8.......
  0x000000b0 01000000 02000000 9f000000 00000000 ................
  0x000000c0 40000000 00000000 01000000 02000000 @...............
  0x000000d0 a5000000 00000000 48000000 00000000 ........H.......
  0x000000e0 01000000 02000000 ab000000 00000000 ................
  0x000000f0 50000000 00000000 01000000 02000000 P...............
  0x00000100 b1000000 00000000 58000000 00000000 ........X.......
  0x00000110 01000000 02000000 b7000000 00000000 ................
  0x00000120 60000000 00000000 01000000 02000000 `...............
  0x00000130 bd000000 00000000 68000000 00000000 ........h.......
  0x00000140 01000000 02000000 c3000000 00000000 ................
  0x00000150 70000000 00000000 01000000 02000000 p...............
  0x00000160 c9000000 00000000 78000000 00000000 ........x.......
  0x00000170 01000000 02000000 cf000000 00000000 ................
  0x00000180 80000000 00000000 01000000 02000000 ................
  0x00000190 d5000000 00000000 88000000 00000000 ................
  0x000001a0 01000000 02000000 db000000 00000000 ................
```

其中的一些数字如 0x69, 0x72, 0x84, 0x8d 正好与下图中 switch 跳转表中跳转到.text 节的偏移量对应，因此我们应该就是要修改这里的数据。



```
重定位节 '.rela.rodata' at offset 0x4e8 contains 26 entries:
  偏移量          信息           类型           符号值        符号名称 + 加数
000000000000  000200000001 R_X86_64_64       0000000000000000 .text + 69
000000000008  000200000001 R_X86_64_64       0000000000000000 .text + 72
000000000010  000200000001 R_X86_64_64       0000000000000000 .text + 7b
000000000018  000200000001 R_X86_64_64       0000000000000000 .text + 84
000000000020  000200000001 R_X86_64_64       0000000000000000 .text + 8d
000000000028  000200000001 R_X86_64_64       0000000000000000 .text + 93
000000000030  000200000001 R_X86_64_64       0000000000000000 .text + 99
```

记住我们所需要的跳转表：
case ‘C’跳转到 c = 0x31
case ‘J’跳转到 c = 0x31
case ‘M’跳转到 c = 0x39
case ‘W’跳转到 c = 0x30
case ‘F’跳转到 c = 0x32
case ‘D’跳转到 c = 0x30
case ‘V’跳转到 c = 0x30
case ‘R’跳转到 c = 0x39
case ‘N’跳转到 c = 0x31
case ‘T’跳转到 c = 0x30
还要.text 节的各个指令偏移：
c=0x31 位于偏移量为 0x84 的位置
c=0x39 位于偏移量为 0x69 的位置
c=0x30 位于偏移量为 0x93 的位置
c=0x32 位于偏移量为 0xf3 的位置

字符’C’是第 3 个字母，因此修改.rela.rodata 节的第 3 个重定位目标使其跳转到.text 节偏移 0x84 处。(7b→84)
字符’J’是第 10 个字母，因此修改.rela.rodata 节的第 10 个重定位目标使其跳转到.text 节偏移 0x84 处。(ab→84)
字符’M’是第 13 个字母，因此修改.rela.rodata 节的第 13 个重定位目标使其跳转到.text 节偏移 0x69 处。(bd→69)
字符’W’是第 23 个字母，因此修改.rela.rodata 节的第 23 个重定位目标使其跳转到.text 节偏移 0x93 处。(f9→93)
字符’F’是第 6 个字母，因此修改.rela.rodata 节的第 6 个重定位目标使其跳转到.text 节偏移 0xf3 处。(93→f3)
字符’D’是第 4 个字母，因此修改.rela.rodata 节的第 4 个重定位目标使其跳转到.text 节偏移 0x93 处。(84→93)
字符’V’是第 22 个字母，因此修改.rela.rodata 节的第 22 个重定位目标使其跳转到.text 节偏移 0x93 处。(f3→93)
字符’R’是第 18 个字母，因此修改.rela.rodata 节的第 18 个重定位目标使其跳转到.text 节偏移 0x69 处。(db→69)
字符’N’是第 14 个字母，因此修改.rela.rodata 节的第 14 个重定位目标使其跳转到.text 节偏移 0x84 处。(c3→84)

字符'T'是第 20 个字母，因此修改.rela.rodata 节的第 20 个重定位目标使其跳转到.text 节偏移 0x93 处。(e7→93)

然后通过 hexedit 编辑这个 ELF。上面分析过.rela.rodata 节位于文件偏移 0x4e8 的位置。

```
000004E0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  69 00 00 00  00 00 00 00
00000500   08 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  72 00 00 00  00 00 00 00  10 00 00 00  00 00 00 00
00000520   01 00 00 00  02 00 00 00  7B 00 00 00  00 00 00 00  18 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
00000540   84 00 00 00  00 00 00 00  20 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  8D 00 00 00  00 00 00 00
00000560   28 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  93 00 00 00  00 00 00 00  30 00 00 00  00 00 00 00
00000580   01 00 00 00  02 00 00 00  99 00 00 00  00 00 00 00  38 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
000005A0   9F 00 00 00  00 00 00 00  40 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  A5 00 00 00  00 00 00 00
000005C0   48 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  AB 00 00 00  00 00 00 00  50 00 00 00  00 00 00 00
000005E0   01 00 00 00  02 00 00 00  B1 00 00 00  00 00 00 00  58 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
00000600   B7 00 00 00  00 00 00 00  60 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  BD 00 00 00  00 00 00 00
00000620   68 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  C3 00 00 00  00 00 00 00  70 00 00 00  00 00 00 00
00000640   01 00 00 00  02 00 00 00  C9 00 00 00  00 00 00 00  78 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
00000660   CF 00 00 00  00 00 00 00  80 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  D5 00 00 00  00 00 00 00
00000680   88 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  DB 00 00 00  00 00 00 00  90 00 00 00  00 00 00 00
000006A0   01 00 00 00  02 00 00 00  E1 00 00 00  00 00 00 00  98 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
000006C0   E7 00 00 00  00 00 00 00  A0 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  ED 00 00 00  00 00 00 00
000006E0   A8 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  F3 00 00 00  00 00 00 00  B0 00 00 00  00 00 00 00
00000700   01 00 00 00  02 00 00 00  F9 00 00 00  00 00 00 00  B8 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
00000720   FF 00 00 00  00 00 00 00  C0 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  05 01 00 00  00 00 00 00
00000740   C8 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  0B 01 00 00  00 00 00 00  20 00 00 00  00 00 00 00
00000760   02 00 00 00  02 00 00 00  00 00 00 00  00 00 00 00  00 2E 73 79  6D 74 61 62  00 2E 73 74  72 74 61 62
00000780   00 2E 73 68  73 74 72 74  61 62 00 2E  72 65 6C 61  2E 74 65 78  74 00 2E 72  65 6C 61 2E  64 61 74 61
000007A0   00 2E 62 73  73 00 2E 72  65 6C 61 2E  72 6F 64 61  74 61 00 2E  63 6F 6D 6D  65 6E 74 00  2E 6E 6F 74
000007C0   65 2E 47 4E  55 2D 73 74  61 63 6B 00  2E 6E 6F 74  65 2E 67 6E  75 2E 70 72  6F 70 65 72  74 79 00 2E
```

上图中标出的是所有偏移量的大小，小端序存储。

我们需要修改其中的 10 个。在下图中用红框标出。

```
000004E0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  69 00 00 00  00 00 00 00
00000500   08 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  72 00 00 00  00 00 00 00  10 00 00 00  00 00 00 00
00000520   01 00 00 00  02 00 00 00  7B 00 00 00  00 00 00 00  18 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
00000540   84 00 00 00  00 00 00 00  20 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  8D 00 00 00  00 00 00 00
00000560   28 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  93 00 00 00  00 00 00 00  30 00 00 00  00 00 00 00
00000580   01 00 00 00  02 00 00 00  99 00 00 00  00 00 00 00  38 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
000005A0   9F 00 00 00  00 00 00 00  40 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  A5 00 00 00  00 00 00 00
000005C0   48 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  AB 00 00 00  00 00 00 00  50 00 00 00  00 00 00 00
000005E0   01 00 00 00  02 00 00 00  B1 00 00 00  00 00 00 00  58 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
00000600   B7 00 00 00  00 00 00 00  60 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  BD 00 00 00  00 00 00 00
00000620   68 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  C3 00 00 00  00 00 00 00  70 00 00 00  00 00 00 00
00000640   01 00 00 00  02 00 00 00  C9 00 00 00  00 00 00 00  78 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
00000660   CF 00 00 00  00 00 00 00  80 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  D5 00 00 00  00 00 00 00
00000680   88 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  DB 00 00 00  00 00 00 00  90 00 00 00  00 00 00 00
000006A0   01 00 00 00  02 00 00 00  E1 00 00 00  00 00 00 00  98 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
000006C0   E7 00 00 00  00 00 00 00  A0 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  ED 00 00 00  00 00 00 00
000006E0   A8 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  F3 00 00 00  00 00 00 00  B0 00 00 00  00 00 00 00
00000700   01 00 00 00  02 00 00 00  F9 00 00 00  00 00 00 00  B8 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
00000720   FF 00 00 00  00 00 00 00  C0 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  05 01 00 00  00 00 00 00
00000740   C8 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  0B 01 00 00  00 00 00 00  20 00 00 00  00 00 00 00
00000760   02 00 00 00  02 00 00 00  00 00 00 00  00 00 00 00  00 2E 73 79  6D 74 61 62  00 2E 73 74  72 74 61 62
00000780   00 2E 73 68  73 74 72 74  61 62 00 2E  72 65 6C 61  2E 74 65 78  74 00 2E 72  65 6C 61 2E  64 61 74 61
```

修改后如下图所示。

```
000004E0   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  69 00 00 00  00 00 00 00
00000500   08 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  72 00 00 00  00 00 00 00  10 00 00 00  00 00 00 00
00000520   01 00 00 00  02 00 00 00  84 00 00 00  00 00 00 00  18 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
00000540   93 00 00 00  00 00 00 00  20 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  8D 00 00 00  00 00 00 00
00000560   28 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  F3 00 00 00  00 00 00 00  30 00 00 00  00 00 00 00
00000580   01 00 00 00  02 00 00 00  99 00 00 00  00 00 00 00  38 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
000005A0   9F 00 00 00  00 00 00 00  40 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  A5 00 00 00  00 00 00 00
000005C0   48 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  84 00 00 00  00 00 00 00  50 00 00 00  00 00 00 00
000005E0   01 00 00 00  02 00 00 00  B1 00 00 00  00 00 00 00  58 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
00000600   B7 00 00 00  00 00 00 00  60 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  69 00 00 00  00 00 00 00
00000620   68 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  84 00 00 00  00 00 00 00  70 00 00 00  00 00 00 00
00000640   01 00 00 00  02 00 00 00  C9 00 00 00  00 00 00 00  78 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
00000660   CF 00 00 00  00 00 00 00  80 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  D5 00 00 00  00 00 00 00
00000680   88 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  69 00 00 00  00 00 00 00  90 00 00 00  00 00 00 00
000006A0   01 00 00 00  02 00 00 00  E1 00 00 00  00 00 00 00  98 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
000006C0   93 00 00 00  00 00 00 00  A0 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  ED 00 00 00  00 00 00 00
000006E0   A8 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  93 00 00 00  00 00 00 00  B0 00 00 00  00 00 00 00
00000700   01 00 00 00  02 00 00 00  93 00 00 00  00 00 00 00  B8 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00
00000720   FF 00 00 00  00 00 00 00  C0 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  05 01 00 00  00 00 00 00
00000740   C8 00 00 00  00 00 00 00  01 00 00 00  02 00 00 00  0B 01 00 00  00 00 00 00  20 00 00 00  00 00 00 00
00000760   02 00 00 00  02 00 00 00  00 00 00 00  00 00 00 00  00 2E 73 79  6D 74 61 62  00 2E 73 74  72 74 61 62
00000780   00 2E 73 68  73 74 72 74  61 62 00 2E  72 65 6C 61  2E 74 65 78  74 00 2E 72  65 6C 61 2E  64 61 74 61
000007A0   00 2E 62 73  73 00 2E 72  65 6C 61 2E  72 6F 64 61  74 61 00 2E  63 6F 6D 6D  65 6E 74 00  2E 6E 6F 74
000007C0   65 2E 47 4E  55 2D 73 74  61 63 6B 00  2E 6E 6F 74  65 2E 67 6E  75 2E 70 72  6F 70 65 72  74 79 00 2E
000007E0   72 65 6C 61  2E 65 68 5F  66 72 61 6D  65 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00
```

然后保存退出。链接并运行程序：

```
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 hexedit phase4.o
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 gcc -no-pie -o phase4_result main.o phase4.o
yx1190200910@icsUbuntu2004 ~/Desktop/ICS/lab5/linklab-1190200910 ./phase4_result
1190200910
```

可以看到正确输出了学号。

3.5 阶段 5 的分析

程序运行结果截图：

分析与设计的过程：

第 4 章 总结

4.1 请总结本次实验的收获

- 深入理解了程序的链接过程
- 学会了 hexedit 软件的使用
- 学会了手动编辑可执行文件的方法(hexedit)，可以用来破解软件

4.2 请给出对本次实验内容的建议

- Phase3 中 PHASE3_CODEBOOK 数组名实际上对于每个学生都不一样，一开始做实验时直接把数组名当成了 PHASE3_CODEBOOK 来做，而不是 MwmaKAcZfZ，最后通过符号表才发现了这个坑，希望老师能多给一些提示。
- Phase5 中不知道如何编写重定位条目的二进制代码，希望老师能给一些提示。

注：本章为酌情加分项。

参考文献

为完成本次实验你翻阅的书籍与网站等

[1] 林来兴. 空间控制技术[M]. 北京：中国宇航出版社，1992：25-42.

[2] 辛希孟. 信息技术与信息服务国际研讨会论文集：A 集[C]. 北京：中国科学出版社，1999.

[3] 赵耀东. 新时代的工业工程师[M/OL]. 台北：天下文化出版社，1998 [1998-09-26]. http://www.ie.nthu.edu.tw/info/ie.newie.htm（Big5）.

[4] 谌颖. 空间交会控制理论与方法研究[D]. 哈尔滨：哈尔滨工业大学，1992：8-13.

[5] KANAMORI H. Shaking Without Quaking[J]. Science，1998，279（5359）：2063-2064.

[6] CHRISTINE M. Plant Physiology: Plant Biology in the Genome Era[J/OL]. Science，1998，281：331-332[1998-09-23]. http://www.sciencemag.org/cgi/collection/anatmorp.

[7] https://baike.baidu.com/item/ELF/7120560?fr=aladdin