哈爾濱工業大學

实验报告

实 验（四）

题　　目 LinkLab

链接

专　　业 计算机类

学　　号 1170500913

班　　级 1703002

学　　生 熊健羽

指 导 教 师 史先俊

实 验 地 点 G712

实 验 日 期 2018.11.12

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

CPU：Intel(R) Core(TM) i5-7200U @ 2.50GHz (64 位)
GPU：Intel(R) HD Graphics 620
Nvidia GeForce 940MX
物理内存：8.00GB
磁盘：1TB HDD
128GB SSD

1.2.2 软件环境

Windows10 64 位；
Vmware 14.11；
Ubuntu 18.04 64 位；

1.2.3 开发工具

Visual Studio 2010 64 位；
Code::Blocks；
gedit，gcc，notepad++；

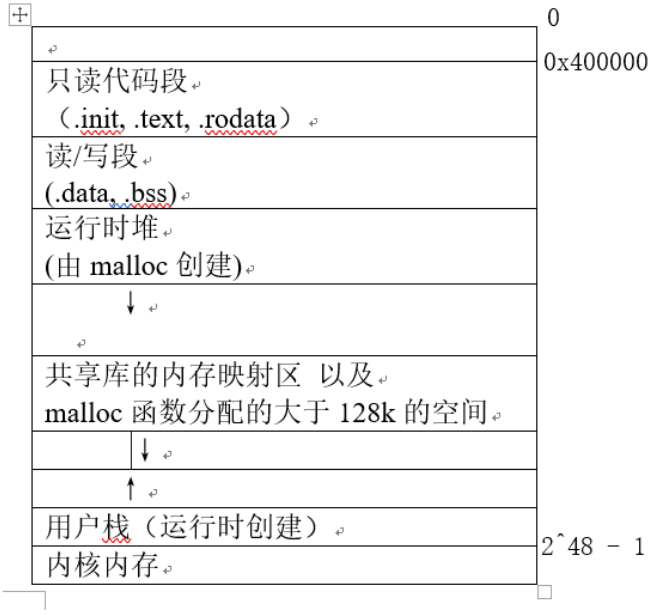
1.3 实验预习

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 头：描述文件总体格式，包括程序的入口点
段头部表：描述可执行文件的节映射到运行时内存段的映射关系
.init：定义了一个小函数，程序的初始化代码会调用它
.text：链接后的代码段
.rodata：链接后的只读数据段
.data：链接后的已初始化的全局/静态变量段
.bss：链接后的未初始化的全局/静态变量或初始化为 0 的全局/静态变量段
.symtab:链接后的符号表
.debug：调试符号表
.line
.strtab
节头部表

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）

1. 内存虚拟内存区：
0x800000000000 - 0xffffffffffff

2. envstring 环境变量字符串：
env[0] *env 0x7ffd0ad2e329 140724785046313
.
.
.
env[54] *env 0x7ffd0ad2efc6 140724785049542
env[55] *env 0x7ffd0ad2efda 140724785049562

3. argv string 命令行字符串：
argv[0] 0x7ffd0ad2e303 140724785046275
./linkaddress
argv[1] 0x7ffd0ad2e311 140724785046289
-u
argv[2] 0x7ffd0ad2e314 140724785046292
1170500913
argv[3] 0x7ffd0ad2e31f 140724785046303
熊健羽

4. env pointers 环境变量指针表：
env 0x7ffd0ad2ded0 140724785045200

5. 命令行参数指针表：
argv 0x7ffd0ad2dea8 140724785045160

6. 命令行参数个数
argc0x7ffd0ad2d97c 140724785043836

7. main 函数的栈帧（因为 local int 为 main 函数里的第一个局部变量）：
local int 0x7ffd0ad2d988 140724785043848
local str 0x7ffd0ad2d9c0 140724785043904

8. 共享函数映射区：
exit 0x7f4e9772a120 139975525048608
printf 0x7f4e9774be80 139975525187200
malloc 0x7f4e9777e070 139975525392496
free 0x7f4e9777e950 139975525394768

strcpy 0x7f4e9779d540 139975525520704

9. 共享内存分配区：（mmap > 128k）
p1 0x7f4e876e6010 139975256334352
p3 0x7f4e97cc3010 139975530917904
p4 0x7f4e476e5010 139974182588432
p5 0x7f4dc76e4010 139972035100688

10. heap 运行时堆 区：
p2 0x555eeb960670 93866167764592

11. .bss 未初始化全局变量区：
big array 0x555ee9fcd060 93866140946528
huge array 0x555ea9fcd060 93865067204704

12. .data 初始化的全局变量区：
global 0x555ea9fcd010 93865067204624
gint 0x555ea9fcd014 93865067204628
glong 0x555ea9fcd018 93865067204632

13. .rodata 只读数据区：
gc 0x555ea9dcbf48 93865065103176
cc 0x555ea9dcbf60 93865065103200

14. .text 只读代码段：
main 0x555ea9dcb8a8 93865065101480
show_pointer 0x555ea9dcb875 93865065101429
useless 0x555ea9dcb86a 93865065101418

15. init 初始化代码段：
init
start

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）

（main 之前）

入口:ld_linux.so id.so.conf

call _dl_start
call dl init(dl-init.c); call init
jmp start

_start:
call __libc_ start main(libc-start.c) halt

__libc_ start main:
call _GI__cxa_atexit (cxa atexitc)
call _libc_csu_init： call init (41B)
call setjmp (bsd-_setjmp.S)
call main

(main 之后)
call _Gl_exit(exit.c)

__GI_exit(exit.c):
call __run_exit_handlers(exitc)

run exit handlers(exitc) :
call _dl_fini
call _IO_cleanup
call __exit: syscall 退出

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：

```
xjy1170500913@ubuntu:~/hitics/lab5/linklab-1170500913$ ./link1
1170500913
```

分析与设计的过程：

使用 gcc -m32 -o linkbomb main.o phase1.o 命令链接生成可执行目标文件，运行得到以下结果：

```
xjy1170500913@ubuntu:~/hitics/lab5/linklab-1170500913$ ./link1_org
9J3V0ulEW 7xRLouZkhYOEEGSIguybmr6OowdXdckG8mqcXHVmmlT5bPTD0k    ZhFh    4v1Rl7GD HePmvbMQia
MzdBiGlIVuQm4tnu9AeIIoPnQZvrWK8k        izT     8M
```

使用 objdump 工具查看 phase1.o 的反汇编码：

```
00000000 <do_phase>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   53                      push   %ebx
   4:   83 ec 04                sub    $0x4,%esp
   7:   e8 fc ff ff ff          call   8 <do_phase+0x8>
                        8: R_386_PC32   __x86.get_pc_thunk.ax
   c:   05 01 00 00 00          add    $0x1,%eax
                        d: R_386_GOTPC  _GLOBAL_OFFSET_TABLE_
  11:   8d 90 44 00 00 00       lea    0x44(%eax),%edx
                        13: R_386_GOTOFF        .data
  17:   83 ec 0c                sub    $0xc,%esp
  1a:   52                      push   %edx
  1b:   89 c3                   mov    %eax,%ebx
  1d:   e8 fc ff ff ff          call   1e <do_phase+0x1e>
                        1e: R_386_PLT32 puts
  22:   83 c4 10                add    $0x10,%esp
  25:   90                      nop
  26:   8b 5d fc                mov    -0x4(%ebp),%ebx
  29:   c9                      leave
  2a:   c3                      ret
```

由图中可知，输出的字符串位于.data 节。

使用 readelf 工具，先查看 phase1.o 的节头表，找到.data 节的偏移：

```
节头：
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .group            GROUP           00000000 000034 000008 04     13  13  4
  [ 2] .text             PROGBITS        00000000 00003c 00002b 00  AX  0   0  1
  [ 3] .rel.text         REL             00000000 000324 000020 08   I 13   2  4
  [ 4] .data             PROGBITS        00000000 000080 0000c1 00  WA  0   0 32
  [ 5] .bss              NOBITS          00000000 000141 000000 00  WA  0   0  1
  [ 6] .data.rel.local   PROGBITS        00000000 000144 000004 00  WA  0   0  4
  [ 7] .rel.data.rel.loc REL             00000000 000344 000008 08   I 13   6  4
  [ 8] .text.__x86.get_p PROGBITS        00000000 000148 000004 00 AXG  0   0  1
  [ 9] .comment          PROGBITS        00000000 00014c 000025 01  MS  0   0  1
  [10] .note.GNU-stack   PROGBITS        00000000 000171 000000 00      0   0  1
  [11] .eh_frame         PROGBITS        00000000 000174 000050 00   A  0   0  4
  [12] .rel.eh_frame     REL             00000000 00034c 000010 08   I 13  11  4
  [13] .symtab           SYMTAB          00000000 0001c4 000110 10     14  12  4
  [14] .strtab           STRTAB          00000000 0002d4 00004d 00      0   0  1
  [15] .shstrtab         STRTAB          00000000 00035c 00008e 00      0   0  1
```

可知.data 节的偏移为 0x80.

于是，使用 hexedit 工具查看 phase1.o 的内容：

```
00000000   7F 45 4C 46  01 01 01 00  00 00 00 00  00 00 00 00  .ELF............
00000010   01 00 03 00  01 00 00 00  00 00 00 00  00 00 00 00  ................
00000020   EC 03 00 00  00 00 00 00  34 00 00 00  00 00 28 00  ........4.....(.
00000030   10 00 0F 00  01 00 00 00  08 00 00 00  55 89 E5 53  ............U..S
00000040   83 EC 04 E8  FC FF FF FF  05 01 00 00  00 8D 90 44  ...............D
00000050   00 00 00 83  EC 0C 52 89  C3 E8 FC FF  FF FF 83 C4  ......R.........
00000060   10 90 8B 5D  FC C9 C3 00  00 00 00 00  00 00 00 00  ...]............
00000070   00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  ................
00000080   55 63 77 77  5A 71 44 6E  31 4B 42 66  6A 36 35 4D  UcwwZqDn1KBfj65M
00000090   53 70 39 42  4E 51 64 46  31 6D 6C 30  55 50 69 4E  Sp9BNQdF1ml0UPiN
000000A0   38 50 4C 59  57 6F 34 44  75 57 33 65  54 34 76 62  8PLYWo4DuW3eT4vb
000000B0   20 51 43 59  70 47 53 48  75 6F 66 30  49 67 6F 44   QCYpGSHuof0IgoD
000000C0   4D 49 78 53  39 4A 33 56  30 75 6C 45  57 20 37 78  MIxS9J3V0ulEW 7x
000000D0   52 4C 6F 75  5A 6B 68 59  4F 45 45 47  53 49 67 75  RLouZkhYOEEGSIgu
000000E0   79 62 6D 72  36 4F 6F 77  64 58 64 63  6B 47 38 6D  ybmr6OowdXdckG8m
000000F0   71 63 58 48  56 6D 6D 6C  54 35 62 50  54 44 30 6B  qcXHVmmlT5bPTD0k
00000100   09 5A 68 46  68 09 34 76  31 52 6C 37  47 44 20 48  .ZhFh.4v1Rl7GD H
00000110   65 50 6D 76  62 4D 51 69  61 4D 7A 64  42 69 47 6C  ePmvbMQiaMzdBiGl
00000120   49 56 75 51  6D 34 74 6E  75 39 41 65  49 49 6F 50  IVuQm4tnu9AeIIoP
00000130   6E 51 5A 76  72 57 4B 38  6B 09 69 7A  54 09 38 4D  nQZvrWK8k.izT.8M
00000140   00 00 00 00  00 00 00 00  8B 04 24 C3  00 47 43 43  ..........$..GCC
00000150   3A 20 28 55  62 75 6E 74  75 20 37 2E  33 2E 30 2D  : (Ubuntu 7.3.0-
00000160   31 36 75 62  75 6E 74 75  33 29 20 37  2E 33 2E 30  16ubuntu3) 7.3.0
```

结合刚才运行结果：

```
xjy1170500913@ubuntu:~/hitics/lab5/linklab-1170500913$ ./link1_org
9J3V0ulEW 7xRLouZkhYOEEGSIguybmr6OowdXdckG8mqcXHVmmlT5bPTD0k    ZhFh    4v1Rl7GD HePmvbMQia
MzdBiGlIVuQm4tnu9AeIIoPnQZvrWK8k        izT     8M
```

把对应的字符串改成我的学号 1170500913，并在后面加上’\0’结束符：

```
00000080   55 63 77 77  5A 71 44 6E  31 4B 42 66  6A 36 35 4D  UcwwZqDn1KBfj65M
00000090   53 70 39 42  4E 51 64 46  31 6D 6C 30  55 50 69 4E  Sp9BNQdF1ml0UPiN
000000A0   38 50 4C 59  57 6F 34 44  75 57 33 65  54 34 76 62  8PLYWo4DuW3eT4vb
000000B0   20 51 43 59  70 47 53 48  75 6F 66 30  49 67 6F 44   QCYpGSHuof0IgoD
000000C0   4D 49 78 53  31 31 37 30  35 30 30 39  31 33 00 78  MIxS1170500913.x
000000D0   52 4C 6F 75  5A 6B 68 59  4F 45 45 47  53 49 67 75  RLouZkhYOEEGSIgu
000000E0   79 62 6D 72  36 4F 6F 77  64 58 64 63  6B 47 38 6D  ybmr6OowdXdckG8m
000000F0   71 63 58 48  56 6D 6D 6C  54 35 62 50  54 44 30 6B  qcXHVmmlT5bPTD0k
00000100   09 5A 68 46  68 09 34 76  31 52 6C 37  47 44 20 48  .ZhFh.4v1Rl7GD H
00000110   65 50 6D 76  62 4D 51 69  61 4D 7A 64  42 69 47 6C  ePmvbMQiaMzdBiGl
00000120   49 56 75 51  6D 34 74 6E  75 39 41 65  49 49 6F 50  IVuQm4tnu9AeIIoP
00000130   6E 51 5A 76  72 57 4B 38  6B 09 69 7A  54 09 38 4D  nQZvrWK8k.izT.8M
00000140   00 00 00 00  00 00 00 00  8B 04 24 C3  00 47 43 43  ..........$..GCC
```

保存，重新链接，运行，成功：

```
xjy1170500913@ubuntu:~/hitics/lab5/linklab-1170500913$ ./link1
1170500913
```

3.2 阶段 2 的分析

程序运行结果截图：

```
xjy1170500913@ubuntu:~/hitics/lab5/linklab-1170500913$ ./link2_new
1170500913
```

分析与设计的过程：

反汇编查看 phase2.o:

```
00000000 <UzPcJtUJ>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   53                      push   %ebx
   4:   83 ec 04                sub    $0x4,%esp
   7:   e8 fc ff ff ff          call   8 <UzPcJtUJ+0x8>
                        8: R_386_PC32   __x86.get_pc_thunk.bx
   c:   81 c3 02 00 00 00       add    $0x2,%ebx
                        e: R_386_GOTPC  _GLOBAL_OFFSET_TABLE_
  12:   83 ec 08                sub    $0x8,%esp
  15:   8d 83 00 00 00 00       lea    0x0(%ebx),%eax
                        17: R_386_GOTOFF        .rodata
  1b:   50                      push   %eax
  1c:   ff 75 08                pushl  0x8(%ebp)
  1f:   e8 fc ff ff ff          call   20 <UzPcJtUJ+0x20>
                        20: R_386_PLT32 strcmp
  24:   83 c4 10                add    $0x10,%esp
  27:   85 c0                   test   %eax,%eax
  29:   75 10                   jne    3b <UzPcJtUJ+0x3b>
  2b:   83 ec 0c                sub    $0xc,%esp
  2e:   ff 75 08                pushl  0x8(%ebp)
  31:   e8 fc ff ff ff          call   32 <UzPcJtUJ+0x32>
                        32: R_386_PLT32 puts
  36:   83 c4 10                add    $0x10,%esp
  39:   eb 01                   jmp    3c <UzPcJtUJ+0x3c>
  3b:   90                      nop
  3c:   8b 5d fc                mov    -0x4(%ebp),%ebx
  3f:   c9                      leave
  40:   c3                      ret
```

分析可知，这块代码是UzPcJtUJ函数引用.rodata段的代码，

再根据 PPT 中的提示：

该函数是这样的：

```
static void OUTPUT_FUNC_NAME( const char *id )    // 该函数名对每名学生均不同
{
    if( strcmp(id,MYID) != 0 ) return;
    printf("%s\n", id);
}
```

只要我们在 do_phase 里以学号为参数，调用该函数即可。

由上述反汇编代码可知，学号”1170500913”存储在.rodata 里。

我们只需编写反汇编代码，使得 do_phase 里以学号为参数，调用该函数即可.

首先查看 do_phase 中现有的汇编代码：

```
00000041 <do_phase>:
  41:   55                      push   %ebp
  42:   89 e5                   mov    %esp,%ebp
  44:   e8 fc ff ff ff          call   45 <do_phase+0x4>
                        45: R_386_PC32  __x86.get_pc_thunk.ax
  49:   05 01 00 00 00          add    $0x1,%eax
                        4a: R_386_GOTPC _GLOBAL_OFFSET_TABLE_
  4e:   90                      nop
  4f:   90                      nop
  50:   90                      nop
  51:   90                      nop
  52:   90                      nop
  53:   90                      nop
  54:   90                      nop
  55:   90                      nop
  56:   90                      nop
  57:   90                      nop
  58:   90                      nop
```

发现红框中的代码，与上面 UzPcJtUJ 函数访问.rodata 处有相似之处。于是想到模仿其操作：

```
00000000 <UzPcJtUJ>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   53                      push   %ebx
   4:   83 ec 04                sub    $0x4,%esp
   7:   e8 fc ff ff ff          call   8 <UzPcJtUJ+0x8>
                        8: R_386_PC32   __x86.get_pc_thunk.bx
   c:   81 c3 02 00 00 00       add    $0x2,%ebx
                        e: R_386_GOTPC  _GLOBAL_OFFSET_TABLE_
  12:   83 ec 08                sub    $0x8,%esp
  15:   8d 83 00 00 00 00       lea    0x0(%ebx),%eax
                        17: R_386_GOTOFF        .rodata
  1b:   50                      push   %eax
```

分析可知，这块代码是UzPcJtUJ函数引用.rodata段的代码，

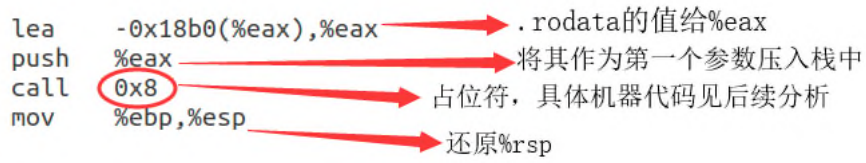
然鹅，其中有一部分为重定位部分：

```
   7:   e8 fc ff ff ff          call   8 <UzPcJtUJ+0x8>
                        8: R_386_PC32   __x86.get_pc_thunk.bx
   c:   81 c3 02 00 00 00       add    $0x2,%ebx
                        e: R_386_GOTPC  _GLOBAL_OFFSET_TABLE_
  12:   83 ec 08                sub    $0x8,%esp
  15:   8d 83 00 00 00 00       lea    0x0(%ebx),%eax
                        17: R_386_GOTOFF        .rodata
```

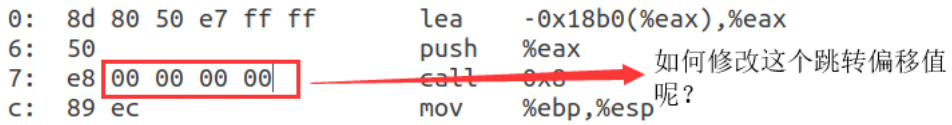
我们还无法得知其真实值。于是想到，先链接出可执行程序，在反汇编出相应汇编代码，从而获得其重定位之后的值：

```
000005b5 <UzPcJtUJ>:
 5b5:   55                      push   %ebp
 5b6:   89 e5                   mov    %esp,%ebp
 5b8:   53                      push   %ebx
 5b9:   83 ec 04                sub    $0x4,%esp
 5bc:   e8 9f fe ff ff          call   460 <__x86.get_pc_thunk.bx>
 5c1:   81 c3 13 1a 00 00       add    $0x1a13,%ebx
 5c7:   83 ec 08                sub    $0x8,%esp
 5ca:   8d 83 50 e7 ff ff       lea    -0x18b0(%ebx),%eax
 5d0:   50                      push   %eax
```

此时，重定位的值已经出现。我们可以模仿其编写汇编代码了：



反汇编得到如下机器代码：



我们打算把代码插入的位置是：

```
00000041 <do_phase>:
  41:   55                      push   %ebp
  42:   89 e5                   mov    %esp,%ebp
  44:   e8 fc ff ff ff          call   45 <do_phase+0x4>
                        45: R_386_PC32  __x86.get_pc_thunk.ax
  49:   05 01 00 00 00          add    $0x1,%eax
                        4a: R_386_GOTPC _GLOBAL_OFFSET_TABLE_
  4e:   90                      nop
  4f:   90                      nop
  50:   90                      nop
  51:   90                      nop
  52:   90                      nop
  53:   90                      nop
  54:   90                      nop
```

而 UzPcJtUJ 函数的相对位置是：

```
00000000 <UzPcJtUJ>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   53                      push   %ebx
   4:   83 ec 04                sub    $0x4,%esp
   7:   e8 fc ff ff ff          call   8 <UzPcJtUJ+0x8>
                        8: R_386_PC32   __x86.get_pc_thunk.bx
   c:   81 c3 02 00 00 00       add    $0x2,%ebx
                        e: R_386_GOTPC  _GLOBAL_OFFSET_TABLE_
  12:   83 ec 08                sub    $0x8,%esp
  15:   8d 83 00 00 00 00       lea    0x0(%ebx),%eax
```

由于 call 指令码后的操作数为 目标地址 – 下条指令地址

目标地址：0

下条指令地址：0x4e + 0xc = 0x5a

所以：操作数 = 0 – 0x5a = 0xFFFFFFA6

所以最终汇编代码应该是：

```
8d 80 50 e7 ff ff
50
e8 a6 ff ff ff
89 ec
```

然后使用 readelf 查看 phase2.o：

```
节头：
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .group            GROUP           00000000 000034 000008 04     16  20  4
  [ 2] .group            GROUP           00000000 00003c 000008 04     16  15  4
  [ 3] .text             PROGBITS        00000000 000044 000071 00  AX  0   0  1
  [ 4] .rel.text         REL             00000000 000344 000038 08   I 16   3  4
  [ 5] .data             PROGBITS        00000000 0000b5 000000 00  WA  0   0  1
  [ 6] .bss              NOBITS          00000000 0000b5 000000 00  WA  0   0  1
  [ 7] .rodata           PROGBITS        00000000 0000b5 00000b 00   A  0   0  1
  [ 8] .data.rel.local   PROGBITS        00000000 0000c0 000004 00  WA  0   0  4
  [ 9] .rel.data.rel.loc REL             00000000 00037c 000008 08   I 16   8  4
  [10] .text.__x86.get_p PROGBITS        00000000 0000c4 000004 00 AXG  0   0  1
  [11] .text.__x86.get_p PROGBITS        00000000 0000c8 000004 00 AXG  0   0  1
  [12] .comment          PROGBITS        00000000 0000cc 000025 01  MS  0   0  1
  [13] .note.GNU-stack   PROGBITS        00000000 0000f1 000000 00      0   0  1
  [14] .eh_frame         PROGBITS        00000000 0000f4 000084 00   A  0   0  4
  [15] .rel.eh_frame     REL             00000000 000384 000020 08   I 16  14  4
  [16] .symtab           SYMTAB          00000000 000178 000160 10     17  15  4
  [17] .strtab           STRTAB          00000000 0002d8 00006a 00      0   0  1
  [18] .shstrtab         STRTAB          00000000 0003a4 0000b2 00      0   0  1
```

确定代码段偏移为 0x44，于是使用 hexedit 编辑：

```
00000000   7F 45 4C 46  01 01 01 00  00 00 00 00  00 00 00 00  .ELF............
00000010   01 00 03 00  01 00 00 00  00 00 00 00  00 00 00 00  ................
00000020   58 04 00 00  00 00 00 00  34 00 00 00  00 00 28 00  X.......4.....(.
00000030   13 00 12 00  01 00 00 00  0A 00 00 00  01 00 00 00  ................
00000040   0B 00 00 00  55 89 E5 53  83 EC 04 E8  FC FF FF FF  ....U..S........
00000050   81 C3 02 00  00 00 83 EC  08 8D 83 00  00 00 00 50  ...............P
00000060   FF 75 08 E8  FC FF FF FF  83 C4 10 85  C0 75 10 83  .u...........u..
00000070   EC 0C FF 75  08 E8 FC FF  FF FF 83 C4  10 EB 01 90  ...u............
00000080   8B 5D FC C9  C3 55 89 E5  E8 FC FF FF  FF 05 01 00  .]...U..........
00000090   00 00 8D 80  50 E7 FF FF  50 E8 A6 FF  FF FF 89 EC  ....P...P.......
000000A0   90 90 90 90  90 90 90 90  90 90 90 90  90 90 90 90  ................
000000B0   90 90 90 5D  C3 31 31 37  30 35 30 30  39 31 33 00  ...].1170500913.
000000C0   00 00 00 00  8B 04 24 C3  8B 1C 24 C3  00 47 43 43  ......$...$..GCC
000000D0   3A 20 28 55  62 75 6E 74  75 20 37 2E  33 2E 30 2D  : (Ubuntu 7.3.0-
000000E0   31 36 75 62  75 6E 74 75  33 29 20 37  2E 33 2E 30  16ubuntu3) 7.3.0
000000F0   00 00 00 00  14 00 00 00  00 00 00 00  01 7A 52 00  .............zR.
```

将代码插入到上图红线所示的正确位置，保存。重新链接，运行，成功：

```
xjy1170500913@ubuntu:~/hitics/lab5/linklab-1170500913$ ./link2_new
1170500913
```

3.3 阶段 3 的分析

程序运行结果截图：

```
xjy1170500913@ubuntu:~/hitics/lab5/linklab-1170500913$ ./link3_new
1170500913
```

分析与设计的过程：

尝试链接并运行，发现只输出一个换行符。

结合 PPT 中内容的解析：

- phase3.c程序框架

```
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" );
}
```

首先要查看 cookie 数组的内容：

使用 objdump 工具反汇编 phase_3.o：

分析下列反汇编代码可知：

```
00000000 <do_phase>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   53                      push   %ebx
   4:   83 ec 24                sub    $0x24,%esp
   7:   e8 fc ff ff ff          call   8 <do_phase+0x8>
                        8: R_386_PC32   __x86.get_pc_thunk.bx
   c:   81 c3 02 00 00 00       add    $0x2,%ebx
                        e: R_386_GOTPC  _GLOBAL_OFFSET_TABLE_
  12:   65 a1 14 00 00 00       mov    %gs:0x14,%eax
  18:   89 45 f4                mov    %eax,-0xc(%ebp)
  1b:   31 c0                   xor    %eax,%eax
  1d:   c7 45 e9 62 6d 64 66    movl   $0x66646d62,-0x17(%ebp)
  24:   c7 45 ed 63 70 6e 79    movl   $0x796e7063,-0x13(%ebp)
  2b:   66 c7 45 f1 77 76       movw   $0x7677,-0xf(%ebp)
  31:   c6 45 f3 00             movb   $0x0,-0xd(%ebp)
  35:   c7 45 e4 00 00 00 00    movl   $0x0,-0x1c(%ebp)
  3c:   eb 2b                   jmp    69 <do_phase+0x69>
  3e:   8d 55 e9                lea    -0x17(%ebp),%edx
  41:   8b 45 e4                mov    -0x1c(%ebp),%eax
  44:   01 d0                   add    %edx,%eax
  46:   0f b6 00                movzbl (%eax),%eax
  49:   0f b6 c0                movzbl %al,%eax
  4c:   8b 93 00 00 00 00       mov    0x0(%ebx),%edx
                        4e: R_386_GOT32X        hWIlnufsMu
  52:   0f b6 04 02             movzbl (%edx,%eax,1),%eax
  56:   0f be c0                movsbl %al,%eax
  59:   83 ec 0c                sub    $0xc,%esp
  5c:   50                      push   %eax
  5d:   e8 fc ff ff ff          call   5e <do_phase+0x5e>
                        5e: R_386_PLT32 putchar
  62:   83 c4 10                add    $0x10,%esp
  65:   83 45 e4 01             addl   $0x1,-0x1c(%ebp)
  69:   8b 45 e4                mov    -0x1c(%ebp),%eax
  6c:   83 f8 09                cmp    $0x9,%eax
  6f:   76 cd                   jbe    3e <do_phase+0x3e>
```

cookie数组首地址

计数变量i

全局数组首地址

为了查看运行时-0x17(%ebp)内存里的值，使用 gdb 运行可执行目标文件，运行到对应位置，使用(gdb) x/10ub $ebp – 0x17 命令查看 cookie 数组的值：

```
(gdb) x/10ub $ebp-0x17
0xffffd021:     98      109     100     102     99      112     110     121
0xffffd029:     119     118
```

因此，我们要使得，全局变量数组的相应下标的元素，为我的学号中的数字：

即：

PHASE3_CODEBOOK[98] = ‘1’

PHASE3_CODEBOOK[109] = ‘1’

PHASE3_CODEBOOK[100] = ‘7’

PHASE3_CODEBOOK[102] = ‘0’

PHASE3_CODEBOOK[99] = ‘5’

PHASE3_CODEBOOK[112] = ‘0’

PHASE3_CODEBOOK[110] = ‘0’

PHASE3_CODEBOOK[121] = ‘9’

PHASE3_CODEBOOK[119] = ‘1’

PHASE3_CODEBOOK[118] = ‘3’

由于在 phase3.o 中，PHASE3_CODEBOOK 数组是未初始化的全局变量，在符号解析中属于弱符号，所以，我们在 phase3_patch.o 中，只需定义同名的全局变量，并初始化为我们想要的数值即可。由于是强符号，符号解析时会优先选择它。

从上面的反汇编代码中，可知该全局数组的符号名为：hWIlnufsMu

```
49:   0f b6 c0                movzbl %al,%eax
4c:   8b 93 00 00 00 00       mov    0x0(%ebx),%edx
                      4e: R_386_GOT32X        hWIlnufsMu
52:   0f b6 04 02             movzbl (%edx,%eax,1),%eax
56:   0f be c0                movsbl %al,%eax
59:   83 ec 0c                sub    $0xc,%esp
```

于是，编写 phase3_patch.c 文件如下：

```
char hWIlnufsMu[256] = {
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '1', '5', '7', '#', '0', '#', '#', '#', '#', '#', '#', '1', '0', '#',
'0', '#', '#', '#', '#', '#', '3', '1', '#', '9', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'1', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#',
'#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#', '#'
};
```

编译为.o 文件，再与 phase3.o main.o 链接得到可执行文件，运行，成功：

```
xjy1170500913@ubuntu:~/hitics/lab5/linklab-1170500913$ gcc -m32 -c phase3_patch.c
xjy1170500913@ubuntu:~/hitics/lab5/linklab-1170500913$ gcc -m32 phase3_patch.o main.o phase
3.o -o link3_new
xjy1170500913@ubuntu:~/hitics/lab5/linklab-1170500913$ ./link3_new
1170500913
```

3.4 阶段 4 的分析

程序运行结果截图：

```
xjy1170500913@ubuntu:~/桌面/hitics/lab5/add$ ./link4
1170500913
```

分析与设计的过程：

ppt 上提供的函数框架：

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);
    }
}
```

首先得查看 cookie 字符串：

反汇编 phase4.o 得：

```
Disassembly of section .text:

00000000 <do_phase>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   53                      push   %ebx
   4:   83 ec 24                sub    $0x24,%esp
   7:   e8 fc ff ff ff          call   8 <do_phase+0x8>
                        8: R_386_PC32   __x86.get_pc_thunk.bx
   c:   81 c3 02 00 00 00       add    $0x2,%ebx
                        e: R_386_GOTPC  _GLOBAL_OFFSET_TABLE_
  12:   65 a1 14 00 00 00       mov    %gs:0x14,%eax
  18:   89 45 f4                mov    %eax,-0xc(%ebp)
  1b:   31 c0                   xor    %eax,%eax
  1d:   c7 45 e9 58 42 48 44    movl   $0x44484258,-0x17(%ebp)
  24:   c7 45 ed 51 46 55 4d    movl   $0x4d554651,-0x13(%ebp)
  2b:   66 c7 45 f1 5a 47       movw   $0x475a,-0xf(%ebp)
  31:   c6 45 f3 00             movb   $0x0,-0xd(%ebp)
  35:   c7 45 e4 00 00 00 00    movl   $0x0,-0x1c(%ebp)
  3c:   e9 e7 00 00 00          jmp    128 <.L30+0x19>
  41:   8d 55 e9                lea    -0x17(%ebp),%edx
  44:   8b 45 e4                mov    -0x1c(%ebp),%eax
  47:   01 d0                   add    %edx,%eax
  49:   0f b6 00                movzbl (%eax),%eax
  4c:   88 45 e3                mov    %al,-0x1d(%ebp)
  4f:   0f be 45 e3             movsbl -0x1d(%ebp),%eax
  53:   83 e8 41                sub    $0x41,%eax
  56:   83 f8 19                cmp    $0x19,%eax
  59:   0f 87 b5 00 00 00       ja     114 <.L30+0x5>
  5f:   c1 e0 02                shl    $0x2,%eax
  62:   8b 84 18 00 00 00 00    mov    0x0(%eax,%ebx,1),%eax
                        65: R_386_GOTOFF        .rodata
  69:   01 d8                   add    %ebx,%eax
  6b:   ff e0                   jmp    *%eax

0000006d <.L4>:
  6d:   c6 45 e3 4a             movb   $0x4a,-0x1d(%ebp)
  71:   e9 9e 00 00 00          jmp    114 <.L30+0x5>
```

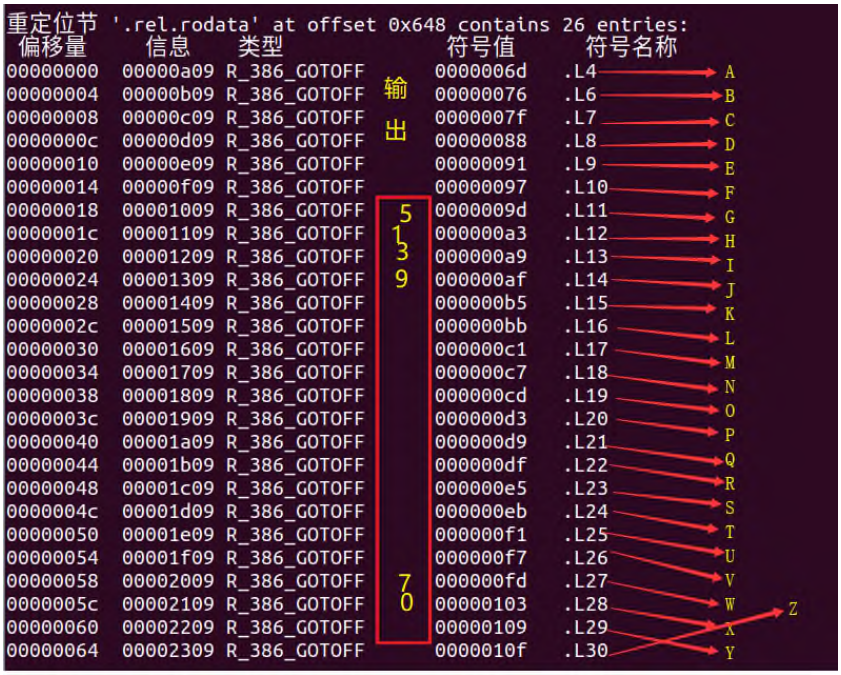
分析可知，-0x17(%ebp)是cookies字符串的首地址

switch部分

使用 gdb，运行至对应位置，查看 cookies 字符串的内容

```
(gdb) x/10cb $ebp-0x17
0xffffd011:     88 'X'  66 'B'  72 'H'  68 'D'  81 'Q'  70 'F'  85 'U'  77 'M'
0xffffd019:     90 'Z'  71 'G'
```

再使用 readelf 工具，查看跳转表：



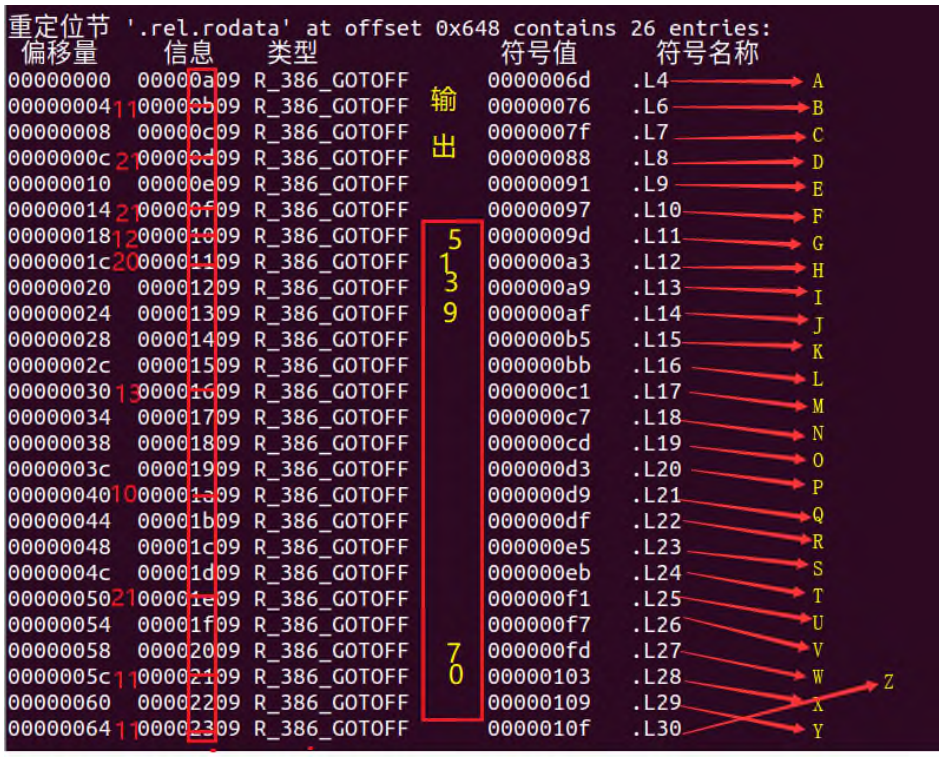
上图的分析中，右侧是相应 case ‘#’ (#为某字母)对应的跳转表项。

我们最后需要输出学号 1170500913，用到的字符为 1 3 5 7 9 0 于是查看 case 语句中的执行内容：

以下是输出字符为 1 3 5 7 9 0 的语句：

```
00000103 <.L28>:
 103:   c6 45 e3 30             movb   $0x30,-0x1d(%ebp)
 107:   eb 0b                   jmp    114 <.L30+0x5>

000000a3 <.L12>:
  a3:   c6 45 e3 31             movb   $0x31,-0x1d(%ebp)
  a7:   eb 6b                   jmp    114 <.L30+0x5>

000000a9 <.L13>:
  a9:   c6 45 e3 33             movb   $0x33,-0x1d(%ebp)
  ad:   eb 65                   jmp    114 <.L30+0x5>

0000009d <.L11>:
  9d:   c6 45 e3 35             movb   $0x35,-0x1d(%ebp)
  a1:   eb 71                   jmp    114 <.L30+0x5>

000000fd <.L27>:
  fd:   c6 45 e3 37             movb   $0x37,-0x1d(%ebp)
 101:   eb 11                   jmp    114 <.L30+0x5>

000000af <.L14>:
  af:   c6 45 e3 39             movb   $0x39,-0x1d(%ebp)
  b3:   eb 5f                   jmp    114 <.L30+0x5>
```

接下来我们要做的就是，修改跳转表项，使得跳转目标变为上述输出字符为 1 3 5 7 9 0 的语句：



readelf 查看.rel.rodata 的位置：

```
[Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
[ 0]                   NULL            00000000 000000 000000 00      0   0  0
[ 1] .group            GROUP           00000000 000034 000008 04     15  39  4
[ 2] .text             PROGBITS        00000000 00003c 000158 00  AX  0   0  1
[ 3] .rel.text         REL             00000000 000618 000030 08   I 15   2  4
[ 4] .data             PROGBITS        00000000 000194 000000 00  WA  0   0  1
[ 5] .bss              NOBITS          00000000 000194 000000 00  WA  0   0  1
[ 6] .rodata           PROGBITS        00000000 000194 000068 00   A  0   0  4
[ 7] .rel.rodata       REL             00000000 000648 0000d0 08   I 15   6  4
[ 8] .data.rel.local   PROGBITS        00000000 0001fc 000004 00  WA  0   0  4
[ 9] .rel.data.rel.loc REL             00000000 000718 000008 08   I 15   8  4
[10] .text.__x86.get_p PROGBITS        00000000 000200 000004 00 AXG  0   0  1
[11] .comment          PROGBITS        00000000 000204 000025 01  MS  0   0  1
```

使用 hexedit 编辑：

```
00000640   4F 01 00 00  02 2A 00 00  00 00 00 00  09 0A 00 00  O....*..........
00000650   04 00 00 00  09 11 00 00  08 00 00 00  09 0C 00 00  ................
00000660   0C 00 00 00  09 21 00 00  10 00 00 00  09 0E 00 00  .....!..........
00000670   14 00 00 00  09 21 00 00  18 00 00 00  09 12 00 00  .....!..........
00000680   1C 00 00 00  09 20 00 00  20 00 00 00  09 12 00 00  ..... .. .......
00000690   24 00 00 00  09 13 00 00  28 00 00 00  09 14 00 00  $.......(.......
000006A0   2C 00 00 00  09 15 00 00  30 00 00 00  09 13 00 00  ,.......0.......
000006B0   34 00 00 00  09 17 00 00  38 00 00 00  09 18 00 00  4.......8.......
000006C0   3C 00 00 00  09 19 00 00  40 00 00 00  09 10 00 00  <.......@.......
000006D0   44 00 00 00  09 1B 00 00  48 00 00 00  09 1C 00 00  D.......H.......
000006E0   4C 00 00 00  09 1D 00 00  50 00 00 00  09 21 00 00  L.......P....!..
000006F0   54 00 00 00  09 1F 00 00  58 00 00 00  09 20 00 00  T.......X.... ..
00000700   5C 00 00 00  09 11 00 00  60 00 00 00  09 22 00 00  \.......`....".."
00000710   64 00 00 00  09 11 00 00  00 00 00 00  01 26 00 00  d............&..
00000720   20 00 00 00  02 02 00 00  44 00 00 00  02 07 00 00   .......D.......
```

保存，重新链接，运行，成功。

```
xjy1170500913@ubuntu:~/桌面/hitics/lab5/add$ ./link4
1170500913
```

3.5 阶段 5 的分析

程序运行结果截图：

```
xjy1170500913@ubuntu:~/桌面/hitics/lab5/phase5$ ./link5
ff4     f
```

分析与设计的过程：

首先查看 phase5.o 的重定位条目.rel.data:

```
重定位节 '.rel.text' at offset 0x6c8 contains 27 entries:
 偏移量     信息    类型              符号值      符号名称
00000000  00000000 R_386_NONE
00000009  00001b0a R_386_GOTPC       00000000   _GLOBAL_OFFSET_TABLE_
00000000  00000000 R_386_NONE
00000025  00000509 R_386_GOTOFF      00000000   .rodata
00000038  00001509 R_386_GOTOFF      00000000   OvZIno
0000004c  00001509 R_386_GOTOFF      00000000   OvZIno
0000005d  00001509 R_386_GOTOFF      00000000   OvZIno
0000006f  00001509 R_386_GOTOFF      00000000   OvZIno
00000000  00000000 R_386_NONE
00000000  00000000 R_386_NONE
00000000  00000000 R_386_NONE
000000a7  00001809 R_386_GOTOFF      0000000b   CODE
00000000  00000000 R_386_NONE
000000c3  00001902 R_386_PC32        00000000   transform_code
000000cc  00001809 R_386_GOTOFF      0000000b   CODE
000000ea  00001d02 R_386_PC32        00000000   __x86.get_pc_thunk.bx
00000000  00000000 R_386_NONE
000000fb  00001f04 R_386_PLT32       00000000   strlen
00000120  00001609 R_386_GOTOFF      00000020   rDAiFt
00000000  00000000 R_386_NONE
00000000  00000000 R_386_NONE
00000189  00001b0a R_386_GOTPC       00000000   _GLOBAL_OFFSET_TABLE_
00000193  00001c02 R_386_PC32        00000090   generate_code
0000019f  00001709 R_386_GOTOFF      00000000   BUF
000001a5  00001e02 R_386_PC32        000000e2   encode
000001b1  00001709 R_386_GOTOFF      00000000   BUF
000001b7  00002104 R_386_PLT32       00000000   puts
```

通过以上条目，可知，被清空了 9 个重定位条目。同时，根据表项的顺序，也可以得知这些条目与已知条目的相对位置。

于是，查看反汇编代码，结合 PPT 中提供的代码框架：

phase5.c程序框架

```
const int TRAN_ARRAY[] = {... ...};
const char FDICT[] = FDICTDAT;
char BUF[] = MYID;
char CODE = PHASE5_COOKIE;

int transform_code( int code, int mode ) {
   switch( TRAN_ARRAY [mode] & 0x00000007 ) {
      case 0:
         code = code & (~ TRAN_ARRAY[mode]);
         break;
      case 1:
         code = code ^ TRAN_ARRAY[mode];
         break;
      ... ...
   }
   return code;
}

void generate_code( int cookie ) {
   int i;
   CODE = cookie;
   for( i=0; i<sizeof(TRAN_ARRAY)/sizeof(int); i++ )
      CODE = transform_code( CODE, i );
}
```

```
int encode( char* str ) {
   int i, n = strlen(str);
   for( i=0; i<n; i++ ) {
      str[i] = (FDICT[str[i]] ^ CODE) & 0x7F;
      if( str[i]<0x20 || str[i]>0x7E ) str[i] = ' ';
   }
   return n;
}

void do_phase() {
   generate_code(PHASE5_COOKIE);
   encode(BUF);
   printf("%s\n", BUF);
}
```

- **上列绿色标出（以及如switch的跳转表等）的符号引用的对应重定位记录中随机选择若干个被置为全零。**
- **涉及的重定位记录可能位于.text, .rodata等不同重定位节中**

21

可以找出 9 个被清除的重定位条目的位置、重定位符号，如下：

```
  1:   89 e5                    mov    %esp,%ebp
  3:   e8 fc ff ff ff           call   4 <transform_code+0x4>
////////////////////// 4:       __x86.get_pc_thunk.ax
  8:   05 01 00 00 00           add    $0x1,%eax
                       9: R_386_GOTPC   _GLOBAL_OFFSET_TABLE_
```

```
  d:   8b 55 0c                 mov    0xc(%ebp),%edx
 10:   8b 94 90 00 00 00 00     mov    0x0(%eax,%edx,4),%edx
//////////////////////13:       OvZIno
 17:   83 e2 07                 and    $0x7,%edx
 1a:   83 fa 07                 cmp    $0x7,%edx
```

```
00000078 <.L9>:
  78:   8b 55 0c                mov    0xc(%ebp),%edx
  7b:   8b 84 90 00 00 00 00    mov    0x0(%eax,%edx,4),%eax
//////////////////////7e:    OvZIno
  82:   89 45 08                mov    %eax,0x8(%ebp)
  85:   eb 04                   jmp    8b <.L2+0x4>
```

```
  8f:   c3                      ret

00000090 <generate_code>:
  90:   55                      push   %ebp
  91:   89 e5                   mov    %esp,%ebp
  93:   53                      push   %ebx
  94:   83 ec 10                sub    $0x10,%esp
  97:   e8 fc ff ff ff          call   98 <generate_code+0x8>
//////////////////////98:    __x86.get_pc_thunk.bx
  9c:   81 c3 02 00 00 00       add    $0x2,%ebx
//////////////////////9d:     _GLOBAL_OFFSET_TABLE_
  a2:   8b 45 08                mov    0x8(%ebp),%eax
  a5:   88 83 00 00 00 00       mov    %al,0x0(%ebx)
                      a7: R_386_GOTOFF        CODE
  ab:   c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%ebp)
  b2:   eb 20                   jmp    d4 <generate_code+0x44>
  b4:   0f b6 83 00 00 00 00    movzbl 0x0(%ebx),%eax
 ////////////////////b7:     CODE
  bb:   0f be c0                movsbl %al,%eax
```

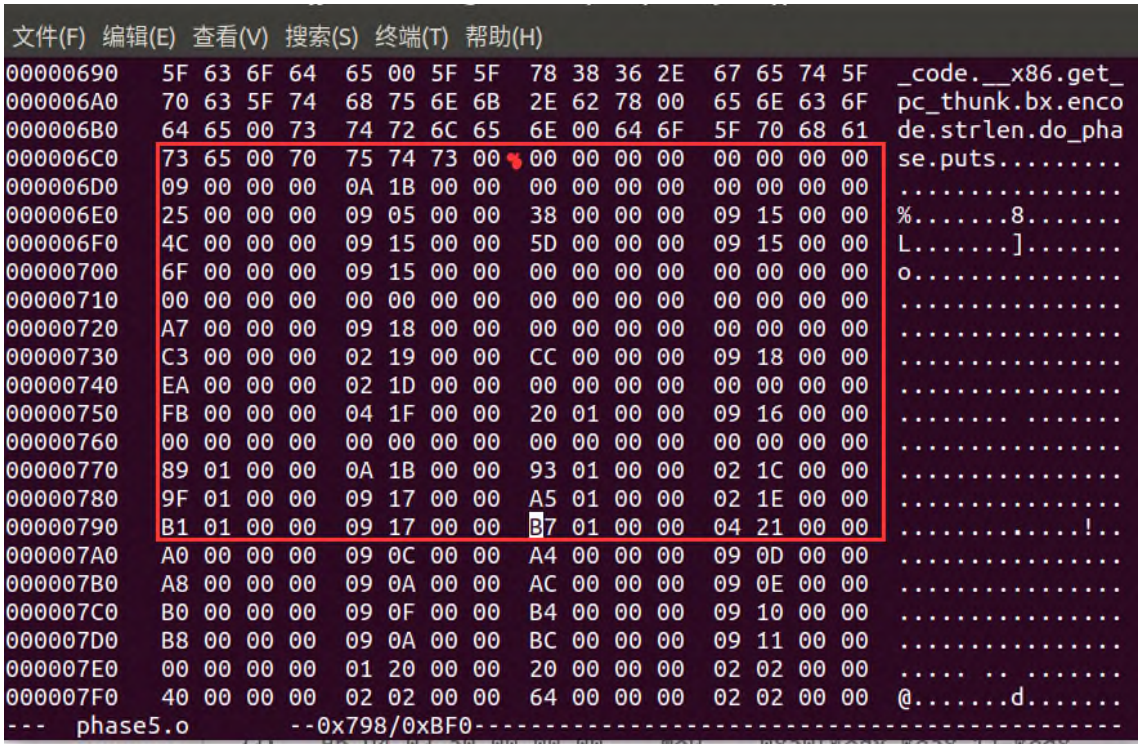
```
000000e2 <encode>:
  e2:   55                      push   %ebp
  e3:   89 e5                   mov    %esp,%ebp
  e5:   53                      push   %ebx
  e6:   83 ec 14                sub    $0x14,%esp
  e9:   e8 fc ff ff ff          call   ea <encode+0x8>
                        ea: R_386_PC32  __x86.get_pc_thunk.bx
  ee:   81 c3 02 00 00 00       add    $0x2,%ebx
////////////////////////f0:   _GLOBAL_OFFSET_TABLE_
  f4:   83 ec 0c                sub    $0xc,%esp
  f7:   ff 75 08                pushl  0x8(%ebp)
  fa:   e8 fc ff ff ff          call   fb <encode+0x19>
                        fb: R_386_PLT32 strlen
10e:   8b 55 f0                 mov    -0x10(%ebp),%edx
111:   8b 45 08                 mov    0x8(%ebp),%eax
114:   01 d0                    add    %edx,%eax
116:   0f b6 00                 movzbl (%eax),%eax
119:   0f be c0                 movsbl %al,%eax
11c:   0f b6 94 03 00 00 00     movzbl 0x0(%ebx,%eax,1),%edx
123:   00
                        120: R_386_GOTOFF       rDAiFt
124:   0f b6 83 00 00 00 00     movzbl 0x0(%ebx),%eax
////////////////////127:     CODE
12b:   89 d1                    mov    %edx,%ecx
12d:   31 c1                    xor    %eax,%ecx

0000017b <do_phase>:
 17b:   55                      push   %ebp
 17c:   89 e5                   mov    %esp,%ebp
 17e:   53                      push   %ebx
 17f:   83 ec 04                sub    $0x4,%esp
 182:   e8 fc ff ff ff          call   183 <do_phase+0x8>
////////////////////////183:    __x86.get_pc_thunk.bx
 187:   81 c3 02 00 00 00       add    $0x2,%ebx
                        189: R_386_GOTPC        _GLOBAL_OFFSET_TABLE_
```

现在，只需把这些重定位条目加入 phase5.o 中即可。

使用 readelf，查看.rel.text 的位置：

```
[Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
[ 0]                   NULL            00000000 000000 000000 00      0   0  0
[ 1] .group            GROUP           00000000 000034 000008 04     17  26  4
[ 2] .group            GROUP           00000000 00003c 000008 04     17  29  4
[ 3] .text             PROGBITS        00000000 000044 0001c4 00  AX  0   0  1
[ 4] .rel.text         REL             00000000 0006c8 0000d8 08   I 17   3  4
[ 5] .data             PROGBITS        00000000 000208 00000c 00  WA  0   0  4
[ 6] .bss              NOBITS          00000000 000214 000000 00  WA  0   0  1
[ 7] .rodata           PROGBITS        00000000 000220 0000c0 00   A  0   0 32
```

使用 edithex，查看偏移为 0x6c8 的位置：



分析可知，每个条目占 8 个字节。

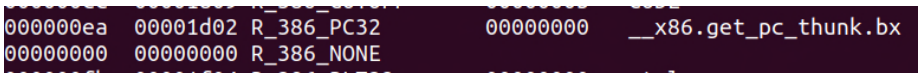
前四个字节：偏移量（大端表示）

后四个字节：符号信息（大端表示）

只需模仿其他已知的条目，照相应的葫芦画瓢，填入即可。

这里有一个问题：需重定位的符号__x86.get_pc_thunk.ax，没有已知重定位条目与之对应。这就需要我们的推断。已知__x86.get_pc_thunk.bx 的 info 为 0x1d02，

发现同时函数符号的 generate_code 的 info 为 0x1c02，**于是我大胆猜想，**__x86.get_pc_thunk.ax 的 info 也是 0xXX02 形式。



查看符号表：

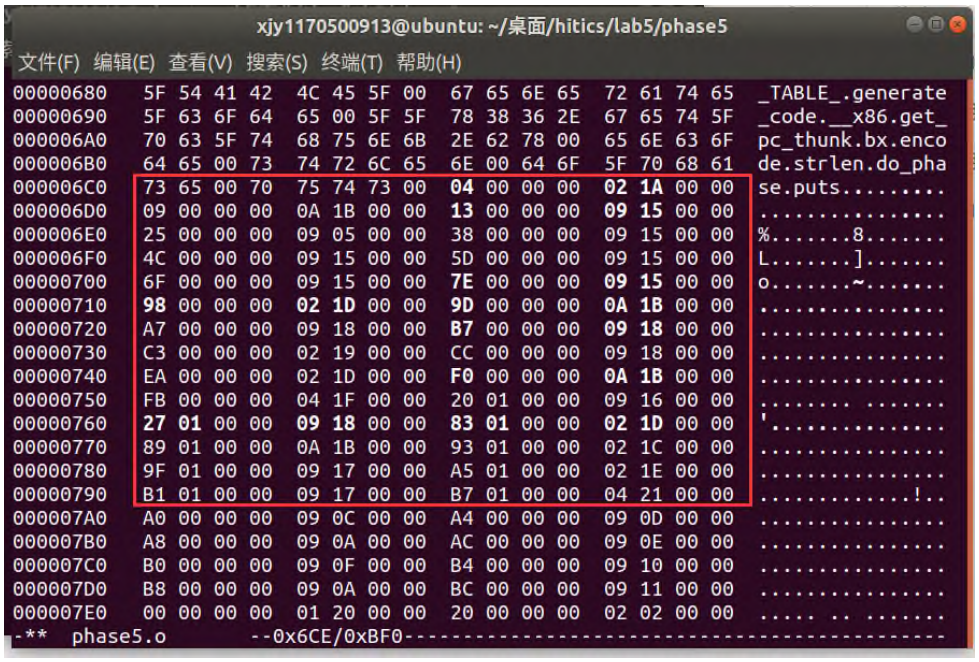
```
18: 00000000     0 SECTION LOCAL  DEFAULT   15
19: 00000000     0 SECTION LOCAL  DEFAULT    1
20: 00000000     0 SECTION LOCAL  DEFAULT    2
21: 00000000    32 OBJECT  GLOBAL DEFAULT    7 OvZIno
22: 00000020   128 OBJECT  GLOBAL DEFAULT    7 rDAiFt
23: 00000000    11 OBJECT  GLOBAL DEFAULT    5 BUF
24: 0000000b     1 OBJECT  GLOBAL DEFAULT    5 CODE
25: 00000000   144 FUNC    GLOBAL DEFAULT    3 transform_code
26: 00000000     0 FUNC    GLOBAL HIDDEN    11 __x86.get_pc_thunk.ax
27: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND _GLOBAL_OFFSET_TABLE_
28: 00000090    82 FUNC    GLOBAL DEFAULT    3 generate_code
29: 00000000     0 FUNC    GLOBAL HIDDEN    12 __x86.get_pc_thunk.bx
30: 000000e2   153 FUNC    GLOBAL DEFAULT    3 encode
31: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND strlen
32: 0000017b    73 FUNC    GLOBAL DEFAULT    3 do_phase
33: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
34: 00000000     4 OBJECT  GLOBAL DEFAULT    9 phase
```

generate_code 的 info 为 0x1c02，而对应的符号表序号为 28，对应的十六进制正好是 0x1c；同样，__x86.get_pc_thunk.bx 的 info 为 0x1d02，而对应的符号表序号为 29，对应的十六进制正好为 0x1d。

据此，由于__x86.get_pc_thunk.ax 对应的符号表序号为 26，对应的十六进制为 0x1a，所以我推断出，__x86.get_pc_thunk.ax 的 info 为 0x1a02.

这样，我们就得知了所有待添加重定位条目的 offset 和 info，就可以开始修改二进制文件了：

修改后的 rel.text 的二进制文件节如下：



现在，再使用 readelf 重新查看 phase5.o 的 rel.text 段：

```
重定位节 '.rel.text' at offset 0x6c8 contains 27 entries:
 偏移量     信息    类型              符号值      符号名称
00000004  00001a02 R_386_PC32        00000000   __x86.get_pc_thunk.ax
00000009  00001b0a R_386_GOTPC       00000000   _GLOBAL_OFFSET_TABLE_
00000013  00001509 R_386_GOTOFF      00000000   OvZIno
00000025  00000509 R_386_GOTOFF      00000000   .rodata
00000038  00001509 R_386_GOTOFF      00000000   OvZIno
0000004c  00001509 R_386_GOTOFF      00000000   OvZIno
0000005d  00001509 R_386_GOTOFF      00000000   OvZIno
0000006f  00001509 R_386_GOTOFF      00000000   OvZIno
0000007e  00001509 R_386_GOTOFF      00000000   OvZIno
00000098  00001d02 R_386_PC32        00000000   __x86.get_pc_thunk.bx
0000009e  00001b0a R_386_GOTPC       00000000   _GLOBAL_OFFSET_TABLE_
000000a7  00001809 R_386_GOTOFF      0000000b   CODE
000000b7  00001809 R_386_GOTOFF      0000000b   CODE
000000c3  00001902 R_386_PC32        00000000   transform_code
000000cc  00001809 R_386_GOTOFF      0000000b   CODE
000000ea  00001d02 R_386_PC32        00000000   __x86.get_pc_thunk.bx
000000f0  00001b0a R_386_GOTPC       00000000   _GLOBAL_OFFSET_TABLE_
000000fb  00001f04 R_386_PLT32       00000000   strlen
00000120  00001609 R_386_GOTOFF      00000020   rDAiFt
00000127  00001809 R_386_GOTOFF      0000000b   CODE
00000183  00001d02 R_386_PC32        00000000   __x86.get_pc_thunk.bx
00000189  00001b0a R_386_GOTPC       00000000   _GLOBAL_OFFSET_TABLE_
00000193  00001c02 R_386_PC32        00000090   generate_code
0000019f  00001709 R_386_GOTOFF      00000000   BUF
000001a5  00001e02 R_386_PC32        000000e2   encode
000001b1  00001709 R_386_GOTOFF      00000000   BUF
000001b7  00002104 R_386_PLT32       00000000   puts
```

可以看到，相应的重定位条目已经成功添加上去了。

此时，重新链接、运行，成功输出编码字符：

```
xjy1170500913@ubuntu:~/桌面/hitics/lab5/phase5$ ./link5
ff4     f
```

第 4 章 总结

4.1 请总结本次实验的收获

很多看书没有看懂的概念，在实验过程中得到了理解。比如重定位条目、符号表、节头部表等概念。

4.2 请给出对本次实验内容的建议

暂无。

注：本章为酌情加分项。

参考文献